TURING 图灵程序设计丛书

MANNING



通过示例全面讲解Java 8新特性

为Java程序员开启函数式编程的大门

Java 8 实战

Java 8 in Action

Lambdas, streams, and functional-style programming

[英] Raoul-Gabriel Urma
[意] Mario Fusco 著
[英] Alan Mycroft
陆明刚 劳佳 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Raoul-Gabriel Urma

剑桥大学计算机科学博士，软件工程师，演讲者，培训师，Cambridge Coding Academy联合创始人、CEO。曾与谷歌、eBay、甲骨文和高盛集团等大公司合作，并参与过多个创业项目。撰写过十余篇经同行审阅的技术文章，并在国际会议上发表过40多篇演讲。

Mario Fusco

Red Hat高级软件工程师，负责JBoss规则引擎Drools的核心开发。拥有丰富的Java开发经验，曾领导媒体公司、金融部门等多个行业的企业级项目开发。对函数式编程和领域特定语言等有浓厚兴趣，并创建了开放源码库lambdaj。

Alan Mycroft

剑桥大学计算机实验室计算学教授，剑桥大学罗宾逊学院研究员，欧洲编程语言和系统协会联合创始人，树莓派基金会联合创始人和理事。发表过大约100篇研究论文，指导过20多篇博士论文。他的研究主要关注编程语言及其语义、优化和实施。他与业界联系紧密，曾于学术休假期间在AT&T实验室和英特尔工作，还创立了Codemist公司，该公司设计了最初的ARM C编译器Norcroft。



陆明刚

毕业于四川大学，目前在EMC中国卓越研发集团任首席工程师，曾任趋势科技中国软件研发中心技术经理，在信息科学和工程领域有十余年的实践和研究经验，拥有多项中国及美国专利。关注JVM性能调优和大数据及其实践，喜欢挖掘技术背后的内幕并乐此不疲。



劳佳

硕士毕业于上海交通大学，现在SAP美国任高级软件支持顾问。业余爱好语言、数学、设计，近年翻译出版了《咨询的奥秘》《卓越程序员密码》等书。

图灵程序设计丛书



Java 8 实战

Java 8 in Action

Lambdas, streams, and functional-style programming

[英] Raoul-Gabriel Urma
[意] Mario Fusco 著
[英] Alan Mycroft
陆明刚 劳佳 译

人民邮电出版社
北 京

图书在版编目（C I P）数据

Java 8实战 / （英）厄马（Urma,R.G.），（意）弗斯
科（Fusco,M.），（英）米克罗夫特（Mycroft,A.）著；
陆明刚，劳佳译. -- 北京 : 人民邮电出版社，2016.4（2016.11重印）
（图灵程序设计丛书）
ISBN 978-7-115-41934-7

Ⅰ. ①J… Ⅱ. ①厄… ②弗… ③米… ④陆… ⑤劳…
Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2016)第047943号

内 容 提 要

本书全面介绍了 Java 8 这个里程碑版本的新特性，包括 Lambdas、流和函数式编程。有了函数式的编程特性，可以让代码更简洁，同时也能自动化地利用多核硬件。全书分四个部分：基础知识、函数式数据处理、高效 Java 8 编程和超越 Java 8，清晰明了地向读者展现了一幅 Java 与时俱进的现代化画卷。

本书适合广大 Java 开发人员阅读。

◆ 著　[英] Raoul-Gabriel Urma [意] Mario Fusco
[英] Alan Mycroft
译　陆明刚　劳　佳
责任编辑　岳新欣
执行编辑　郭亚维
责任印制　彭志环
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：23
字数：552千字　2016年 4 月第 1 版
印数：4 501 – 5 500册　2016年 11 月北京第 3 次印刷
著作权合同登记号　图字：01-2016-1573号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版 权 声 明

谨以此书献给我们的父母。

序 言

1998年，八岁的我拿起了我此生第一本计算机书，那本书讲的是JavaScript和HTML。我当时怎么也想不到，打开那本书会让我见识编程语言和它们能够创造的神奇世界，并会彻底改变我的生活。我被它深深地吸引了。如今，编程语言的某个新特性还会时不时地让我感到兴奋，因为它让我花更少的时间就能够写出更清晰、更简洁的代码。我希望本书探讨的Java 8中那些来自函数式编程的新思想，同样能够给你启迪。

那么，你可能会问，这本书是怎么来的呢?

2011年，甲骨文公司的Java语言架构师Brian Goetz分享了一些在Java中添加Lambda表达式的提议，以期获得业界的参与。这让我重新燃起了兴趣，于是我开始传播这些想法，在各种开发人员会议上组织Java 8讨论班，并为剑桥大学的学生开设讲座。

到了2013年4月，消息不胫而走，Manning出版社的编辑给我发了封邮件，问我是否有兴趣写一本书关于Java 8中Lambda的书。当时我只是个“不起眼”的二年级博士生，似乎写书并不是一个好主意，因为它会耽误我提交论文。另一方面，所谓“只争朝夕”，我想写一本小书不会有太多工作量，对吧？（后来我才意识到自己大错特错！）于是我咨询我的博士生导师Alan Mycroft教授，结果他十分支持我写书（甚至愿意为这种与博士学位无关的工作提供帮助，我永远感谢他）。几天后，我们见到了Java 8的布道者Mario Fusco，他有着非常丰富的专业经验，并且因在重大开发者会议上所做的函数式编程演讲而享有盛名。

我们很快就认识到，如果将大家的能量和背景融合起来，就不仅仅可以写出一本关于Java 8的Lambda的小书，而是可以写出（我们希望）一本五年或十年后，在Java领域仍然有人愿意阅读的书。我们有了一个非常难得的机会来深入讨论许多话题，它们不但有益于Java程序员，还打开了通往一个新世界的大门：函数式编程。

15个月后，到2014年7月，在经历无数个漫漫长夜的辛苦工作、无数次的编辑和永生难忘的体验后，我们的工作成果终于送到了你的手上。希望你会喜欢它！

Raoul-Gabriel Urma
于剑桥大学

致　　谢

如果没有许多杰出人士的支持，这本书是不可能完成的。

- 自愿提供宝贵审稿建议的朋友：Richard Walker、Jan Saganowski、Brian Goetz、Stuart Marks、Cem Redif、Paul Sandoz、Stephen Colebourne、Íñigo Mediavilla、Allahbaksh Asadullah、Tomasz Nurkiewicz和Michael Müller。
- 曼宁早期访问项目（Manning Early Access Program，MEAP）中在作者在线论坛上发表评论的读者。
- 在编撰过程中提供有益反馈的审阅者：Antonio Magnaghi、Brent Stains、Franziska Meyer、Furkan Kamachi、Jason Lee、Jörn Dinkla、Lochana Menikarachchi、Mayur Patil、Nikolaos Kaintantzis、Simone Bordet、Steve Rogers、Will Hayworth和William Wheeler。
- Manning的开发编辑Susan Conant耐心回答了我们所有的问题和疑虑，并为每一章的初稿提供了详尽的反馈，并尽其所能支持我们。
- Ivan Todorović和Jean-François Morin在本书付印前进行了全面的技术审阅，Al Scherer则在编撰过程中提供了技术帮助。

Raoul-Gabriel Urma

首先，我要感谢我的父母在生活中给予我无尽的爱和支持。我写一本书的小小梦想如今成真了！其次，我要向信任并且支持我的博士生导师和合著者Alan Mycroft表达无尽的感激。我也要感谢合著者Mario Fusco陪我走过这段有趣的旅程。最后，我要感谢在生活中为我提供指导、有用建议，给予我鼓励的朋友们：Sophia Drossopoulou、Aidan Roche、Warris Bokhari、Alex Buckley、Martijn Verburg、Tomas Petricek和Tian Zhao。你们真是太棒啦！

Mario Fusco

我要特别感谢我的妻子Marilena，她无尽的耐心让我可以专注于写作本书；还有我们的女儿Sofia，因为她能够创造无尽的混乱，让我可以从本书的写作中暂时抽身。你在阅读本书时将发现，Sofia还用只有两岁小女孩才会的方式，告诉我们内部迭代和外部迭代之间的差异。我还要感谢Raoul-Gabriel Urma和Alan Mycroft，他们与我一起分享了写作本书的（巨大）喜悦和（小小）痛苦。

Alan Mycroft

我要感谢我的太太Hilary和其他家庭成员在本书写作期间对我的忍受，我常常说“再稍微弄弄就好了”，结果一弄就是好几个小时。我还要感谢多年来的同事和学生，他们让我知道了怎么去教授知识。最后，感谢Mario和Raoul这两位非常高效的合著者，特别是Raoul在苛求“周五再交出一部分稿件”时，还能让人愉快地接受。

关于本书

简单地说，Java 8中的新增功能是自Java 1.0发布18年以来，Java发生的最大变化。没有去掉任何东西，因此你现有的Java代码都能工作，但新功能提供了强大的新语汇和新设计模式，能帮助你编写更清楚、更简洁的代码。就像遇到所有新功能时那样，你一开始可能会想："为什么又要去改我的语言呢？"但稍加练习之后，你就会发觉自己只用预期的一半时间，就用新功能写出了更短、更清晰的代码，这时你会意识到自己永远无法返回到"旧Java"了。

本书会帮助你跨过"原理听起来不错，但还是有点儿新，不太适应"的门槛，从而熟练地进行编程。

"也许吧，"你可能会想，"可是Lambda、函数式编程，这些不是那些留着胡子、穿着凉鞋的学究们在象牙塔里面琢磨的东西吗？"或许是的，但Java 8中加入的新想法的分量刚刚好，它们带来的好处也可以被普通的Java程序员所理解。本书会从普通程序员的角度来叙述，偶尔谈谈"这是怎么来的"。

"Lambda，听起来跟天书一样！"是的，也许是这样，但它是一个很好的想法，让你可以编写简明的Java程序。许多人都熟悉事件处理器和回调函数，即注册一个对象，它包含会在事件发生时使用的一个方法。Lambda使人更容易在Java中广泛应用这种思想。简单来说，Lambda和它的朋友"方法引用"让你在做其他事情的过程中，可以简明地将代码或方法作为参数传递进去执行。在本书中，你会看到这种思想出现得比预想的还要频繁：从加入作比较的代码来简单地参数化一个排序方法，到利用新的Stream API在一组数据上表达复杂的查询指令。

"流（stream）是什么？"这是Java 8的一个新功能。它们的特点和集合（collection）差不多，但有几个明显的优点，让我们可以使用新的编程风格。首先，如果你使用过SQL等数据库查询语言，就会发现用几行代码写出的查询语句要是换成Java要写好长。Java 8的流支持这种简明的数据库查询式编程——但用的是Java语法，而无需了解数据库！其次，流被设计成无需同时将所有的数据调入内存（甚至根本无需计算），这样就可以处理无法装入计算机内存的流数据了。但Java 8可以对流做一些集合所不能的优化操作，例如，它可以将对同一个流的若干操作组合起来，从而只遍历一次数据，而不是花很大代价去多次遍历它。更妙的是，Java可以自动将流操作并行化（集合可不行）。

"还有函数式编程，这又是什么？"就像面向对象编程一样，它是另一种编程风格，其核心是把函数作为值，前面在讨论Lambda的时候提到过。

Java 8的好处在于，它把函数式编程中一些最好的想法融入到了大家熟悉的Java语法中。有了这个优秀的设计选择，你可以把函数式编程看作Java 8中一个额外的设计模式和语汇，让你可以用更少的时间，编写更清楚、更简洁的代码。想想你的编程兵器库中的利器又多了一样。

当然，除了这些在概念上对Java有很大扩充的功能，我们也会解释很多其他有用的Java 8功能和更新，如默认方法、新的`Optional`类、`CompletableFuture`，以及新的日期和时间API。

别急，这只是一个概览，现在该让你自己去看看本书了。

本书结构

本书分为四个部分："基础知识""函数式数据处理""高效Java 8编程"和"超越Java 8"。我们强烈建议你按顺序阅读，因为很多概念都需要前面的章节作为基础。大多数章节都有几个小测验，帮助你学习和掌握这些内容。

第一部分包括3章，旨在帮助你初步使用Java 8。学完这一部分，你将会对Lambda表达式有充分的了解，并可以编写简洁而灵活的代码，能够轻松适应不断变化的需求。

- 在第1章中，我们总结了Java的主要变化（Lambda表达式、方法引用、流和默认方法），并为学习后面的内容做好准备。
- 在第2章中，你将了解行为参数化，这是Java 8非常依赖的一种软件开发模式，也是引入Lambda表达式的主要原因。
- 第3章全面地解释了Lambda表达式和方法引用，每一步都有代码示例和测验。

第二部分仔细讨论了新的Stream API。学完这一部分，你将充分理解流是什么，以及如何在Java应用程序中使用它们来简洁而高效地处理数据集。

- 第4章介绍了流的概念，并解释它们与集合有何异同。
- 第5章详细讨论了表达复杂数据处理查询可以使用的流操作。我们会谈到很多模式，如筛选、切片、查找、匹配、映射和归约。
- 第6章讲到了收集器——Stream API的一个功能，可以让你表达更为复杂的数据处理查询。
- 在第7章中，你将了解流如何得以自动并行执行，并利用多核架构的优势。此外，你还会学到为正确而高效地使用并行流，要避免的若干陷阱。

第三部分探讨了能让你高效使用Java 8并在代码中运用现代语汇的若干内容。

- 第8章探讨了如何利用Java 8的新功能和一些秘诀来改善你现有的代码。此外，该章还探讨了一些重要的软件开发技术，如设计模式、重构、测试和调试。
- 在第9章中，你将了解到默认方法是什么，如何利用它们来以兼容的方式演变API，一些实际的应用模式，以及有效使用默认方法的规则。
- 第10章谈到了新的`java.util.Optional`类，它能让你设计出更好的API，并减少空指针异常。
- 第11章探讨了`CompletableFuture`，它可以让你用声明性方式表达复杂的异步计算，从而让Stream API的设计并行化。

- 第12章探讨了新的日期和时间API，这相对于以前涉及日期和时间时容易出错的API是一大改进。

在本书最后一部分，我们会返回来谈谈怎么用Java编写高效的函数式程序，还会将Java 8的功能和Scala作一比较。

- 第13章是一个完整的函数式编程教程，介绍了一些术语，并解释了如何在Java 8中编写函数式风格的程序。
- 第14章涵盖了更高级的函数式编程技巧，包括高阶函数、科里化、持久化数据结构、延迟列表和模式匹配。你可以把这一章看作一种融合，既有可以用在代码库中的实际技术，也有让你成为更渊博的程序员的学术知识。
- 第15章对比了Java 8的功能与Scala的功能。Scala和Java一样，是一种实施在JVM上的语言，近年来迅速发展，在编程语言生态系统中已经威胁到了Java的一些方面。
- 在第16章我们会回顾这段学习Java 8并慢慢走向函数式编程的历程。此外，我们还会猜测，在Java 8之后，未来可能还有哪些增强和新功能出现。

最后，本书有四个附录，涵盖了与Java 8相关的其他一些话题。附录A总结了本书未讨论的一些Java 8的小特性。附录B概述了Java库的其他主要扩展，可能对你有用。附录C是第二部分的延续，谈到了流的高级用法。附录D探讨了Java编译器在幕后是如何实现Lambda表达式的。

代码惯例和下载

所有代码清单和正文中的源代码都采用等宽字体（如`fixed-widthfontlikethis`），以与普通文字区分开来。许多代码清单中都有注释，突出了重要的概念。

书中所有示例代码和执行说明均可见于https://github.com/java8/Java8InAction。你也可以从出版商网站（https://www.manning.com/java8inaction）下载包含本书所有示例的zip文件。

作者在线

购买本书即可免费访问Manning Publications运营的一个私有在线论坛，你可以在那里发表关于本书的评论、询问技术问题，并获得作者和其他用户的帮助。如欲访问作者在线论坛并订阅，请用浏览器访问https://www.manning.com/java8inaction。这个页面说明了注册后如何使用论坛，能获得什么类型的帮助，以及论坛上的行为守则。

Manning对读者的承诺是提供一个平台，供读者之间以及读者和作者之间进行有意义的对话。但这并不意味着作者会有任何特定程度的参与。他们对论坛的贡献是完全自愿的（且无报酬）。我们建议你试着询问作者一些有挑战性的问题，以免他们失去兴趣！

只要本书仍在印，你就可以在出版商网站上访问作者在线论坛和先前所讨论内容的归档文件。

关于封面图

本书封面上的图为“1700年中国清朝满族战士的服饰”。图片中的人物衣饰华丽，身佩利剑，背背弓和箭筒。如果你仔细看他的腰带，会发现一个λ形的带扣（这是我们的设计师加上去的，暗示本书的主题）。该图选自托马斯·杰弗里斯的《各国古代和现代服饰集》（*A Collection of the Dresses of Different Nations, Ancient and Modern*，伦敦，1757年至1772年间出版），该书标题页中说这些图是手工上色的铜版雕刻品，并且是用阿拉伯树胶填充的。托马斯·杰弗里斯（Thomas Jefferys，1719—1771）被称为“乔治三世的地理学家”。他是一名英国制图员，是当时主要的地图供应商。他为政府和其他官方机构雕刻和印制地图，制作了很多商业地图和地理地图集，尤以北美地区为多。地图制作商的工作让他对勘察和绘图过的地方的服饰产生了兴趣，这些都在这个四卷本中得到了出色的展现。

向往遥远的土地、渴望旅行，在18世纪还是相对新鲜的现象，而类似于这本集子的书籍则十分流行，这些集子向旅游者和坐着扶手椅梦想去旅游的人介绍了其他国家的人。杰弗里斯书中异彩纷呈的图画生动地描绘了几百年前世界各国的独特与个性。如今，着装规则已经改变，各个国家和地区一度非常丰富的多样性也已消失，来自不同大陆的人仅靠衣着已经很难区分开了。不过，要是乐观点儿看，我们这是用文化和视觉上的多样性，换得了更多姿多彩的个人生活——或是更为多样化、更为有趣的知识和技术生活。

计算机书籍一度也是如此繁荣，Manning出版社在此用杰弗里斯画中复活的三个世纪前风格各异的国家服饰，来象征计算机行业中的发明与创造的异彩纷呈。

目　　录

Part 1 第一部分

基础知识

本书第一部分将介绍Java 8的基础知识。学完第一部分，你将会对Lambda表达式有充分的了解，并可以编写简洁而灵活的代码，能够轻松地适应不断变化的需求。

第1章将总结Java的主要变化（Lambda表达式、方法引用、流和默认方法），并为学习本书做好准备。

在第2章中，你将了解行为参数化，这是Java 8非常依赖的一种软件开发模式，也是引入Lambda表达式的主要原因。

第3章全面地解释了Lambda表达式和方法引用的概念，每一步都有代码示例和测验。

第 1 章

为什么要关心Java 8

本章内容

- Java怎么又变了
- 日新月异的计算应用背景：多核和处理大型数据集（大数据）
- 改进的压力：函数式比命令式更适应新的体系架构
- Java 8的核心新特性：Lambda（匿名函数）、流、默认方法

自1998年JDK 1.0（Java 1.0）发布以来，Java已经受到了学生、项目经理和程序员等一大批活跃用户的欢迎。这一语言极富活力，不断被用在大大小小的项目里。从Java 1.1（1997年） 一直到Java 7（2011年），Java通过增加新功能，不断得到良好的升级。Java 8则是在2014年3月发布的。那么，问题来了：为什么你应该关心Java 8?

我们的理由是，Java 8所做的改变，在许多方面比Java历史上任何一次改变都深远。而且好消息是,这些改变会让你编起程来更容易,用不着再写类似下面这种啰嗦的程序了(对inventory中的苹果按照重量进行排序):

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

在Java 8里面，你可以编写更为简洁的代码，这些代码读起来更接近问题的描述：

```
inventory.sort(comparing(Apple::getWeight));
```

本书中第一段Java 8的代码！

它念起来就是“给库存排序，比较苹果的重量”。现在你不用太关注这段代码，本书后面的章节将会介绍它是做什么用的，以及你如何写出类似的代码。

Java 8对硬件也有影响：平常我们用的CPU都是多核的——你的笔记本电脑或台式机上的处理器可能有四个CPU内核，甚至更多。但是，绝大多数现有的Java程序都只使用其中一个内核，其他三个都闲着，或只是用一小部分的处理能力来运行操作系统或杀毒程序。

在Java 8之前，专家们可能会告诉你，必须利用线程才能使用多个内核。问题是，线程用起来很难，也容易出现错误。从Java的演变路径来看，它一直致力于让并发编程更容易、出错更少。

Java 1.0里有线程和锁，甚至有一个内存模型——这是当时的最佳做法，但事实证明，不具备专门知识的项目团队很难可靠地使用这些基本模型。Java 5添加了工业级的构建模块，如线程池和并发集合。Java 7添加了分支/合并（fork/join）框架，使得并行变得更实用，但仍然很困难。而Java 8对并行有了一个更简单的新思路，不过你仍要遵循一些规则，本书中会谈到。

我们用两个例子（它们有更简洁的代码，且更简单地使用了多核处理器）就可以管中窥豹，看到一座拔地而起相互勾连一致的Java 8大厦。首先让你快速了解一下这些想法（希望能引起你的兴趣，也希望我们总结得足够简洁）：

- ❑ Stream API
- ❑ 向方法传递代码的技巧
- ❑ 接口中的默认方法

Java 8提供了一个新的API（称为“流”，Stream），它支持许多处理数据的并行操作，其思路和在数据库查询语言中的思路类似——用更高级的方式表达想要的东西，而由“实现”（在这里是Streams库）来选择最佳低级执行机制。这样就可以避免用`synchronized`编写代码，这一代码不仅容易出错，而且在多核CPU上执行所需的成本也比你想象的要高。[①]

从有点修正主义的角度来看，在Java 8中加入`Streams`可以看作把另外两项扩充加入Java 8的直接原因：*把代码传递给方法的简洁方式*（方法引用、Lambda）和接口中的*默认方法*。

如果仅仅“把代码传递给方法”看作`Streams`的一个结果，那就低估了它在Java 8中的应用范围。它提供了一种新的方式，这种方式简洁地表达了*行为参数化*。比方说，你想要写两个只有几行代码不同的方法，那现在你只需要把不同的那部分代码作为参数传递进去就可以了。采用这种编程技巧，代码会更短、更清晰，也比常用的复制粘贴更不容易出错。高手看到这里就会想，在Java 8之前可以用匿名类实现行为参数化呀——但是想想本章开头那个Java 8代码更加简洁的例子，代码本身就说明了它有多清晰！

Java 8里面将代码传递给方法的功能（同时也能够返回代码并将其包含在数据结构中）还让我们能够使用一整套新技巧，通常称为*函数式编程*。一言以蔽之，这种被函数式编程界称为*函数*的代码，可以被来回传递并加以组合，以产生强大的编程语汇。这样的例子在本书中随处可见。

本章主要从宏观角度探讨了语言为什么会演变，接下来几节介绍Java 8的核心特性，然后介绍函数式编程思想——其新的特性简化了使用，而且更适应新的计算机体系结构。简而言之，1.1节讨论了Java的演变过程和概念，指出Java以前缺乏以简易方式利用多核并行的能力。1.2节介绍了为什么把代码传递给方法在Java 8里是如此强大的一个新的编程语汇。1.3节对`Streams`做了同样的介绍：`Streams`是Java 8表示有序数据，并能灵活地表示这些数据是否可以并行处理的新方式。1.4节解释了如何利用Java 8中的默认方法功能让接口和库的演变更顺畅、编译更少。最后，1.5节展望了在Java和其他共用JVM的语言中进行函数式编程的思想。总的来说，本章会介绍整体脉络，而细节会在本书的其余部分中逐一展开。请尽情享受吧！

---

① 多核CPU的每个处理器内核都有独立的高速缓存。加锁需要这些高速缓存同步运行，然而这又需要在内核间进行较慢的缓存一致性协议通信。

1.1　Java 怎么还在变

20世纪60年代，人们开始追求完美的编程语言。当时著名的计算机科学家彼得·兰丁（Peter Landin）在1966年的一篇标志性论文[①]中写道，当时已经有700种编程语言了，并推测了接下来的700种会是什么样子，文中也对类似于Java 8中的函数式编程进行了讨论。

之后，又出现了数以千计的编程语言。学者们得出结论，编程语言就像生态系统一样，新的语言会出现，旧语言则被取代，除非它们不断演变。我们都希望出现一种完美的通用语言，可在现实中，某些语言只是更适合某些方面。比如，C和C++仍然是构建操作系统和各种嵌入式系统的流行工具，因为它们编出的程序尽管安全性不佳，但运行时占用资源少。缺乏安全性可能导致程序意外崩溃，并把安全漏洞暴露给病毒和其他东西；确实，Java和C#等安全型语言在诸多运行资源不太紧张的应用中已经取代了C和C++。

先抢占市场往往能够吓退竞争对手。为了一个功能而改用新的语言和工具链往往太过痛苦了，但新来者最终会取代现有的语言，除非后者演变得够快，能跟上节奏。年纪大一点的读者大多可以举出一堆这样的语言——他们以前用过，但是现在这些语言已经不时髦了。随便列举几个吧：Ada、Algol、COBOL、Pascal、Delphi、SNOBOL等。

你是一位Java程序员。在过去15年的时间里，Java已经成功地霸占了编程生态系统中的一大块，同时替代了竞争对手语言。让我们来看看其中的原因。

1.1.1　Java 在编程语言生态系统中的位置

Java天资不错。从一开始，它就是一个精心设计的面向对象的语言，有许多有用的库。有了集成的线程和锁的支持，它从第一天起就支持小规模并发（并且它十分有先知之明地承认，在与硬件无关的内存模型里，多核处理器上的并发线程可能比在单核处理器上出现的意外行为更多）。此外，将Java编译成JVM字节码（一种很快就被每一种浏览器支持的虚拟机代码）意味着它成为了互联网applet（小应用）的首选（你还记得applet吗？）。确实，Java虚拟机（JVM）及其字节码可能会变得比Java语言本身更重要，而且对于某些应用来说，Java可能会被同样运行在JVM上的竞争对手语言（如Scala或Groovy）取代。JVM各种最新的更新（例如JDK7中的新`invokedynamic`字节码）旨在帮助这些竞争对手语言在JVM上顺利运行，并与Java交互操作。Java也已成功地占领了嵌入式计算的若干领域，从智能卡、烤面包机、机顶盒到汽车制动系统。

Java是怎么进入通用编程市场的？

面向对象在20世纪90年代开始时兴的原因有两个：封装原则使得其软件工程问题比C少；作为一个思维模型，它轻松地反映了Windows 95及之后的WIMP编程模式。可以这样总结：一切都是对象；单击鼠标就能给处理程序发送一个事件消息（在`Mouse`对象中触发`Clicked`方

① P. J. Landin,“The Next 700 Programming Languages,” *CACM* 9(3):157–65, March 1966.

法)。Java的"一次编写，随处运行"模式，以及早期浏览器安全地执行Java小应用的能力让它占领了大学市场，毕业生随后把它带进了业界。开始时由于运行成本比C/C++要高，Java还遇到了一些阻力，但后来机器变得越来越快，程序员的时间也变得越来越重要了。微软的C#进一步验证了Java的面向对象模型。

但是，编程语言生态系统的气候正在变化。程序员越来越多地要处理所谓的大数据（数百万兆甚至更多字节的数据集），并希望利用多核计算机或计算集群来有效地处理。这意味着需要使用并行处理——Java以前对此并不支持。

你可能接触过其他编程领域的思想，比如Google的map-reduce，或如SQL等数据库查询语言的便捷数据操作，它们能帮助你处理大数据量和多核CPU。图1-1总结了语言生态系统：把这幅图看作编程问题空间，每个特定地方生长的主要植物就是程序最喜欢的语言。气候变化的意思是，新的硬件或新的编程因素（例如，"我为什么不能用SQL的风格来写程序？"）意味着新项目优选的语言各有不同，就像地区气温上升就意味着葡萄在较高的纬度也能长得好。当然这会有滞后——很多老农一直在种植传统作物。总之，新的语言不断出现，并因为迅速适应了气候变化，越来越受欢迎。

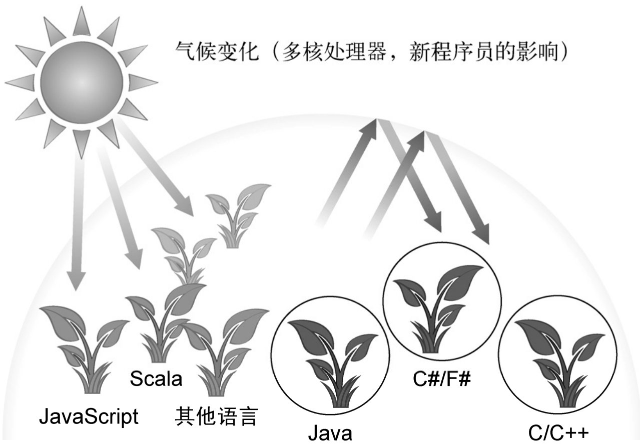


图1-1　编程语言生态系统和气候变化

Java 8对于程序员的主要好处在于它提供了更多的编程工具和概念，能以更快，更重要的是能以更为简洁、更易于维护的方式解决新的或现有的编程问题。虽然这些概念对于Java来说是新的，但是研究型的语言已经证明了它们的强大。我们会突出并探讨三个这样的编程概念背后的思想，它们促使Java 8中开发出并行和编写更简洁通用代码的功能。我们这里介绍它们的顺序和本书其余的部分略有不同，一方面是为了类比Unix，另一方面是为了揭示Java 8新的多核并行中存在的"因为这个所以需要那个"的依赖关系。

1.1.2　流处理

第一个编程概念是流处理。介绍一下，流是一系列数据项，一次只生成一项。程序可以从输入流中一个一个读取数据项，然后以同样的方式将数据项写入输出流。一个程序的输出流很可能是另一个程序的输入流。

一个实际的例子是在Unix或Linux中，很多程序都从标准输入（Unix和C中的stdin，Java中的System.in）读取数据，然后把结果写入标准输出（Unix和C中的stdout，Java中的System.out）。首先我们来看一点点背景：Unix的cat命令会把两个文件连接起来创建一个流，tr会转换流中的字符，sort会对流中的行进行排序，而tail -3则给出流的最后三行。Unix命令行允许这些程序通过管道（|）连接在一起，比如

```
cat file1 file2 | tr "[A-Z]"  "[a-z]"  |  sort  |  tail -3
```

会（假设file1和file2中每行都只有一个词）先把字母转换成小写字母，然后打印出按照词典排序出现在最后的三个单词。我们说sort把一个行流[1]作为输入，产生了另一个行流（进行排序）作为输出，如图1-2所示。请注意在Unix中，命令（cat、tr、sort和tail）是同时执行的，这样sort就可以在cat或tr完成前先处理头几行。就像汽车组装流水线一样，汽车排队进入加工站，每个加工站会接收、修改汽车，然后将之传递给下一站做进一步的处理。尽管流水线实际上是一个序列，但不同加工站的运行一般是并行的。

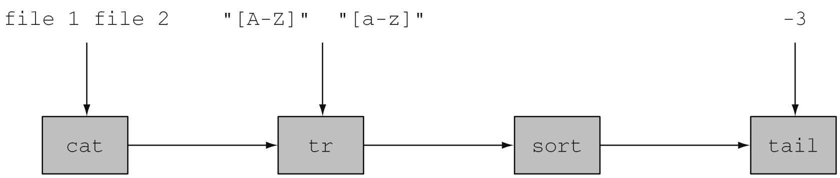


图1-2　操作流的Unix命令

基于这一思想，Java 8在java.util.stream中添加了一个Stream API；Stream<T>就是一系列T类型的项目。你现在可以把它看成一种比较花哨的迭代器。Stream API的很多方法可以链接起来形成一个复杂的流水线，就像先前例子里面链接起来的Unix命令一样。

推动这种做法的关键在于，现在你可以在一个更高的抽象层次上写Java 8程序了：思路变成了把这样的流变成那样的流（就像写数据库查询语句时的那种思路），而不是一次只处理一个项目。另一个好处是，Java 8可以透明地把输入的不相关部分拿到几个CPU内核上去分别执行你的Stream操作流水线——这是几乎免费的并行，用不着去费劲搞Thread了。我们会在第4~7章仔细讨论Java 8的Stream API。

[1] 有语言洁癖的人会说“字符流”，不过认为sort会对行排序比较简单。

1.1.3 用行为参数化把代码传递给方法

Java 8中增加的另一个编程概念是通过API来传递代码的能力。这听起来实在太抽象了。在Unix的例子里，你可能想告诉sort命令使用自定义排序。虽然sort命令支持通过命令行参数来执行各种预定义类型的排序，比如倒序，但这毕竟是有限的。

比方说，你有一堆发票代码，格式类似于2013UK0001、2014US0002……前四位数代表年份，接下来两个字母代表国家，最后四位是客户的代码。你可能想按照年份、客户代码，甚至国家来对发票进行排序。你真正想要的是，能够给sort命令一个参数让用户定义顺序：给sort命令传递一段独立代码。

那么，直接套在Java上，你是要让sort方法利用自定义的顺序进行比较。你可以写一个compareUsingCustomerId来比较两张发票的代码，但是在Java 8之前，你没法把这个方法传给另一个方法。你可以像本章开头时介绍的那样，创建一个Comparator对象，将之传递给sort方法，但这不但啰嗦，而且让“重复使用现有行为”的思想变得不那么清楚了。Java 8增加了把方法（你的代码）作为参数传递给另一个方法的能力。图1-3是基于图1-2画出的，它描绘了这种思路。我们把这一概念称为*行为参数化*。它的重要之处在哪儿呢？Stream API就是构建在通过传递代码使操作行为实现参数化的思想上的，当把compareUsingCustomerId传进去，你就把sort的行为参数化了。

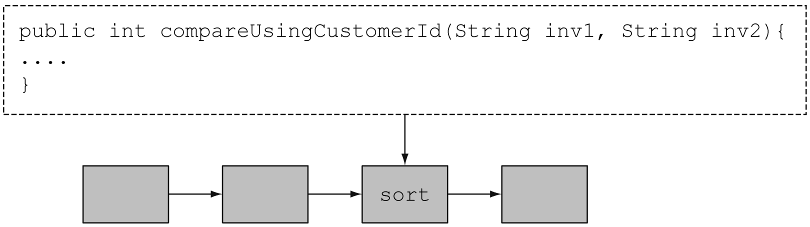


图1-3 将compareUsingCustomerId方法作为参数传给sort

我们将在1.2节中概述这种方式，但详细讨论留在第2章和第3章。第13章和第14章将讨论这一功能的高级用法，还有*函数式编程*自身的一些技巧。

1.1.4 并行与共享的可变数据

第三个编程概念更隐晦一点，它来自我们前面讨论流处理能力时说的“几乎免费的并行”。你需要放弃什么吗？你可能需要对传给流方法的行为的写法稍作改变。这些改变可能一开始会让你感觉有点儿不舒服，但一旦习惯了你就会爱上它们。你的行为必须能够同时对不同的输入安全地执行。一般情况下这就意味着，你写代码时不能访问共享的可变数据。这些函数有时被称为“纯函数”或“无副作用函数”或“无状态函数”，这一点我们会在第7章和第13章详细讨论。前面说的并行只有在假定你的代码的多个副本可以独立工作时才能进行。但如果要写入的是一个共享变量或对象，这就行不通了：如果两个进程需要同时修改这个共享变量怎么办？（1.3节配图给出

了更详细的解释。）你在本书中会对这种风格有更多的了解。

Java 8的流实现并行比Java现有的线程API更容易，因此，尽管可以使用`synchronized`来打破“不能有共享的可变数据”这一规则，但这相当于是在和整个体系作对，因为它使所有围绕这一规则做出的优化都失去意义了。在多个处理器内核之间使用`synchronized`，其代价往往比你预期的要大得多，因为同步迫使代码按照顺序执行，而这与并行处理的宗旨相悖。

这两个要点（没有共享的可变数据，将方法和函数即代码传递给其他方法的能力）是我们平常所说的*函数式编程*范式的基石，我们在第13章和第14章会详细讨论。与此相反，在*命令式编程*范式中，你写的程序则是一系列改变状态的指令。“不能有共享的可变数据”的要求意味着，一个方法是可以通过它将参数值转换为结果的方式完全描述的；换句话说，它的行为就像一个数学函数，没有可见的副作用。

1.1.5　Java 需要演变

你之前已经见过了Java的演变。例如，引入泛型，使用`List<String>`而不只是`List`，可能一开始都挺烦人的。但现在你已经熟悉了这种风格和它所带来的好处，即在编译时能发现更多错误，且代码更易读，因为你现在知道列表里面是什么了。

其他改变让普通的东西更容易表达，比如，使用`for-each`循环而不用暴露`Iterator`里面的套路写法。Java 8中的主要变化反映了它开始远离常侧重改变现有值的经典面向对象思想，而向函数式编程领域转变，在大面上考虑做*什么*（例如，*创建一个值代表所有从A到B低于给定价格的交通线路*）被认为是头等大事，并和*如何*实现（例如，*扫描一个数据结构并修改某些元素*）区分开来。请注意，如果极端点儿来说，传统的面向对象编程和函数式可能看起来是冲突的。但是我们的理念是获得两种编程范式中最好的东西，这样你就有更大的机会为任务找到理想的工具了。我们会在接下来的两节中详细讨论：Java中的函数和新的Stream API。

总结下来可能就是这么一句话：语言需要不断改进以跟进硬件的更新或满足程序员的期待（如果你还不够信服，想想COBOL还一度是商业上最重要的语言之一呢）。要坚持下去，Java必须通过增加新功能来改进，而且只有新功能被人使用，变化才有意义。所以，使用Java 8，你就是在保护你作为Java程序员的职业生涯。除此之外，我们有一种感觉——你一定会喜欢Java 8的新功能。随便问问哪个用过Java 8的人，看看他们愿不愿意退回去。还有，用生态系统打比方的话，新的Java 8的功能使得Java能够征服如今被其他语言占领的编程任务领地，所以Java 8程序员就更需要学习它了。

下面逐一介绍Java 8中的新概念，并顺便指出在哪一章中还会仔细讨论这些概念。

1.2　Java 中的函数

编程语言中的*函数*一词通常是指*方法*，尤其是静态方法；这是在*数学函数*，也就是没有副作用的函数之外的新含义。幸运的是，你将会看到，在Java 8谈到函数时，这两种用法几乎是一致的。

Java 8中新增了函数——值的一种新形式。它有助于使用1.3节中谈到的流，有了它，Java 8可以进行多核处理器上的并行编程。我们首先来展示一下作为值的函数本身的有用之处。

想想Java程序可能操作的值吧。首先有原始值，比如42（int类型）和3.14（double类型）。其次，值可以是对象（更严格地说是对象的引用）。获得对象的唯一途径是利用new，也许是通过工厂方法或库函数实现的；对象引用指向类的一个*实例*。例子包括"abc"（String类型），new Integer(1111)（Integer类型），以及new HashMap<Integer,String>(100)的结果——它显然调用了HashMap的构造函数。甚至数组也是对象。那么有什么问题呢？

为了帮助回答这个问题，我们要注意到，编程语言的整个目的就在于操作值，要是按照历史上编程语言的传统，这些值因此被称为一等值（或一等公民，这个术语是从20世纪60年代美国民权运动中借用来的）。编程语言中的其他结构也许有助于我们表示值的结构，但在程序执行期间不能传递，因而是二等公民。前面所说的值是Java中的一等公民，但其他很多Java概念（如方法和类等）则是二等公民。用方法来定义类很不错，类还可以实例化来产生值，但方法和类本身都不是值。这又有什么关系呢？还真有，人们发现，在运行时传递方法能将方法变成一等公民。这在编程中非常有用，因此Java 8的设计者把这个功能加入到了Java中。顺便说一下，你可能会想，让类等其他二等公民也变成一等公民可能也是个好主意。有很多语言，如Smalltalk和JavaScript，都探索过这条路。

1.2.1 方法和 Lambda 作为一等公民

Scala和Groovy等语言的实践已经证明，让方法等概念作为一等值可以扩充程序员的工具库，从而让编程变得更容易。一旦程序员熟悉了这个强大的功能，他们就再也不愿意使用没有这一功能的语言了。因此，Java 8的设计者决定允许方法作为值，让编程更轻松。此外，让方法作为值也构成了其他若干Java 8功能（如Stream）的基础。

我们介绍的Java 8的第一个新功能是*方法引用*。比方说，你想要筛选一个目录中的所有隐藏文件。你需要编写一个方法，然后给它一个File，它就会告诉你文件是不是隐藏的。幸好，File类里面有一个叫作isHidden的方法。我们可以把它看作一个函数，接受一个File，返回一个布尔值。但要用它做筛选，你需要把它包在一个FileFilter对象里，然后传递给File.listFiles方法，如下所示：

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isHidden();          <—— 筛选隐藏文件
    }
});
```

呃！真可怕！虽然只有三行，但这三行可真够绕的。我们第一次碰到的时候肯定都说过："非得这样不可吗？"我们已经有一个方法isHidden可以使用，为什么非得把它包在一个啰嗦的FileFilter类里面再实例化呢？因为在Java 8之前你必须这么做！

如今在Java 8里，你可以把代码重写成这个样子：

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

哇！酷不酷？你已经有了函数isHidden，因此只需用Java 8的*方法引用*::语法（即“把这个方法作为值”）将其传给listFiles方法；请注意，我们也开始用*函数*代表方法了。稍后我们会解释这个机制是如何工作的。一个好处是，你的代码现在读起来更接近问题的陈述了。方法不再是二等值了。与用*对象引用*传递对象类似（对象引用是用new创建的），在Java 8里写下File::isHidden的时候，你就创建了一个*方法引用*，你同样可以传递它。第3章会详细讨论这一概念。只要方法中有代码（方法中的可执行部分），那么用方法引用就可以传递代码，如图1-3所示。图1-4说明了这一概念。你在下一节中还将看到一个具体的例子——从库存中选择苹果。

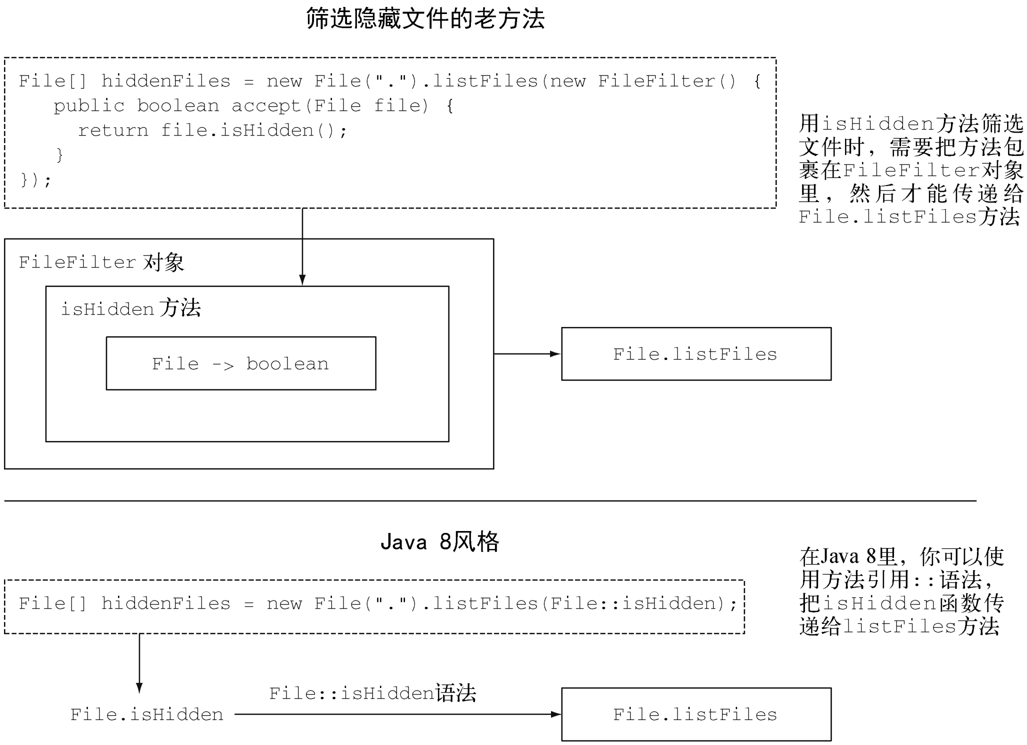


图1-4　将方法引用File::isHidden传递给listFiles方法

Lambda——匿名函数

除了允许（命名）函数成为一等值外，Java 8还体现了更广义的将*函数作为值*的思想，包括Lambda[①]（或匿名函数）。比如，你现在可以写(int x) -> x + 1，表示“调用时给定参数x，就返回x + 1值的函数”。你可能会想这有什么必要呢？因为你可以在MyMathsUtils类里面定义一个add1方法，然后写MyMathsUtils::add1嘛！确实是可以，但要是你没有方便的方法和类

① 最初是根据希腊字母λ命名的。虽然Java中不使用这个符号，名称还是被保留了下来。

可用，新的Lambda语法更简洁。第3章会详细讨论Lambda。我们说使用这些概念的程序为函数式编程风格，这句话的意思是“编写把函数作为一等值来传递的程序”。

1.2.2 传递代码：一个例子

来看一个例子，看看它是如何帮助你写程序的，我们在第2章还会进行更详细的讨论。所有的示例代码均可见于本书的GitHub页面（https://github.com/java8/）。假设你有一个`Apple`类，它有一个`getColor`方法，还有一个变量`inventory`保存着一个`Apples`的列表。你可能想要选出所有的绿苹果，并返回一个列表。通常我们用筛选（filter）一词来表达这个概念。在Java 8之前，你可能会写这样一个方法`filterGreenApples`：

```
public static List<Apple> filterGreenApples(List<Apple> inventory){
    List<Apple> result = new ArrayList<>();          <-- result是用来累积结果的List，开始为空，然后一个个加入绿苹果
    for (Apple apple: inventory){
        if ("green".equals(apple.getColor())) {      <-- 高亮显示的代码会仅仅选出绿苹果
            result.add(apple);
        }
    }
    return result;
}
```

但是接下来，有人可能想要选出重的苹果，比如超过150克，于是你心情沉重地写了下面这个方法，甚至用了复制粘贴：

```
public static List<Apple> filterHeavyApples(List<Apple> inventory){
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {      <-- 这里高亮显示的代码会仅仅选出重的苹果
            result.add(apple);
        }
    }
    return result;
}
```

我们都知道软件工程中复制粘贴的危险——给一个做了更新和修正，却忘了另一个。嘿，这两个方法只有一行不同：`if`里面高亮的那行条件。如果这两个高亮的方法之间的差异仅仅是接受的重量范围不同，那么你只要把接受的重量上下限作为参数传递给`filter`就行了，比如指定`(150, 1000)`来选出重的苹果（超过150克），或者指定`(0, 80)`来选出轻的苹果（低于80克）。

但是，我们前面提过了，Java 8会把条件代码作为参数传递进去，这样可以避免`filter`方法出现重复的代码。现在你可以写：

```
public static boolean isGreenApple(Apple apple) {
    return "green".equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}
public interface Predicate<T>{          <-- 写出来是为了清晰（平常只要从java.util.function导入就可以了）
```

```
    boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

方法作为Predicate参数p传递进去（见附注栏“什么是谓词？”）

苹果符合p所代表的条件吗

要用它的话，你可以写：

```
filterApples(inventory, Apple::isGreenApple);
```

或者

```
filterApples(inventory, Apple::isHeavyApple);
```

我们会在接下来的两章中详细讨论它是怎么工作的。现在重要的是你可以在Java 8里面传递方法了！

> **什么是谓词？**
>
> 前面的代码传递了方法Apple::isGreenApple（它接受参数Apple并返回一个boolean）给filterApples，后者则希望接受一个Predicate<Apple>参数。**谓词**（predicate）在数学上常常用来代表一个类似函数的东西，它接受一个参数值，并返回true或false。你在后面会看到，Java 8也会允许你写Function<Apple,Boolean>——在学校学过函数却没学过谓词的读者对此可能更熟悉，但用Predicate<Apple>是更标准的方式，效率也会更高一点儿，这避免了把boolean封装在Boolean里面。

1.2.3　从传递方法到 Lambda

把方法作为值来传递显然很有用，但要是为类似于isHeavyApple和isGreenApple这种可能只用一两次的短方法写一堆定义有点儿烦人。不过Java 8也解决了这个问题，它引入了一套新记法（匿名函数或Lambda），让你可以写

```
filterApples(inventory, (Apple a) -> "green".equals(a.getColor()) );
```

或者

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

甚至

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     "brown".equals(a.getColor()) );
```

所以，你甚至都不需要为只用一次的方法写定义；代码更干净、更清晰，因为你用不着去找自己到底传递了什么代码。但要是Lambda的长度多于几行（它的行为也不是一目了然）的话，那你还是应该用方法引用来指向一个有描述性名称的方法，而不是使用匿名的Lambda。你应该以代码的清晰度为准绳。

Java 8的设计师几乎可以就此打住了，要是没有多核CPU，可能他们真的就到此为止了。我们迄今为止谈到的函数式编程竟然如此强大，在后面你更会体会到这一点。本来，Java加上`filter`和几个相关的东西作为通用库方法就足以让人满意了，比如

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

这样你甚至都不需要写`filterApples`了，因为比如先前的调用

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

就可以直接调用库方法`filter`：

```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

但是，为了更好地利用并行，Java的设计师没有这么做。Java 8中有一整套新的类集合API——Stream，它有一套函数式程序员熟悉的、类似于`filter`的操作，比如`map`、`reduce`，还有我们接下来要讨论的在`Collections`和`Streams`之间做转换的方法。

1.3 流

几乎每个Java应用都会制造和处理集合。但集合用起来并不总是那么理想。比方说，你需要从一个列表中筛选金额较高的交易，然后按货币分组。你需要写一大堆套路化的代码来实现这个数据处理命令，如下所示：

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();
for (Transaction transaction : transactions) {
    if(transaction.getPrice() > 1000){
        Currency currency = transaction.getCurrency();
        List<Transaction> transactionsForCurrency =
            transactionsByCurrencies.get(currency);
        if (transactionsForCurrency == null) {
            transactionsForCurrency = new ArrayList<>();
            transactionsByCurrencies.put(currency,
                                         transactionsForCurrency);
        }
        transactionsForCurrency.add(transaction);
    }
}
```

建立累积交易分组的**Map**

遍历交易的**List**

筛选金额较高的交易

提取交易货币

如果这个货币的分组**Map**是空的，那就建立一个

将当前遍历的交易添加到具有同一货币的交易**List**中

此外，我们很难一眼看出来这些代码是做什么的，因为有好几个嵌套的控制流指令。

有了Stream API，你现在可以这样解决这个问题了：

```
import static java.util.stream.Collectors.toList;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
                .filter((Transaction t) -> t.getPrice() > 1000)  <-- 筛选金额较高的交易
                .collect(groupingBy(Transaction::getCurrency));   <-- 按货币分组
```

这看起来有点儿神奇，不过现在先不用担心。第4~7章会专门讲述怎么理解Stream API。现在值得注意的是，和Collection API相比，Stream API处理数据的方式非常不同。用集合的话，你得自己去做迭代的过程。你得用for-each循环一个个去迭代元素，然后再处理元素。我们把这种数据迭代的方法称为*外部迭代*。相反，有了Stream API，你根本用不着操心循环的事情。数据处理完全是在库内部进行的。我们把这种思想叫作*内部迭代*。在第4章我们还会谈到这些思想。

使用集合的另一个头疼的地方是，想想看，要是你的交易量非常庞大，你要怎么处理这个巨大的列表呢？单个CPU根本搞不定这么大量的数据，但你很可能已经有了一台多核电脑。理想的情况下，你可能想让这些CPU内核共同分担处理工作，以缩短处理时间。理论上来说，要是你有八个核，那并行起来，处理数据的速度应该是单核的八倍。

多核

所有新的台式和笔记本电脑都是多核的。它们不是仅有一个CPU，而是有四个、八个，甚至更多CPU，通常称为内核[①]。问题是，经典的Java程序只能利用其中一个核，其他核的处理能力都浪费了。类似地，很多公司利用**计算集群**（用高速网络连接起来的多台计算机）来高效处理海量数据。Java 8提供了新的编程风格，可更好地利用这样的计算机。

Google的搜索引擎就是一个无法在单台计算机上运行的代码的例子。它要读取互联网上的每个页面并建立索引，将每个互联网网页上出现的每个词都映射到包含该词的网址上。然后，如果你用多个单词进行搜索，软件就可以快速利用索引，给你一个包含这些词的网页集合。想想看，你会如何在Java中实现这个算法，哪怕是比Google小的引擎也需要你利用计算机上所有的核。

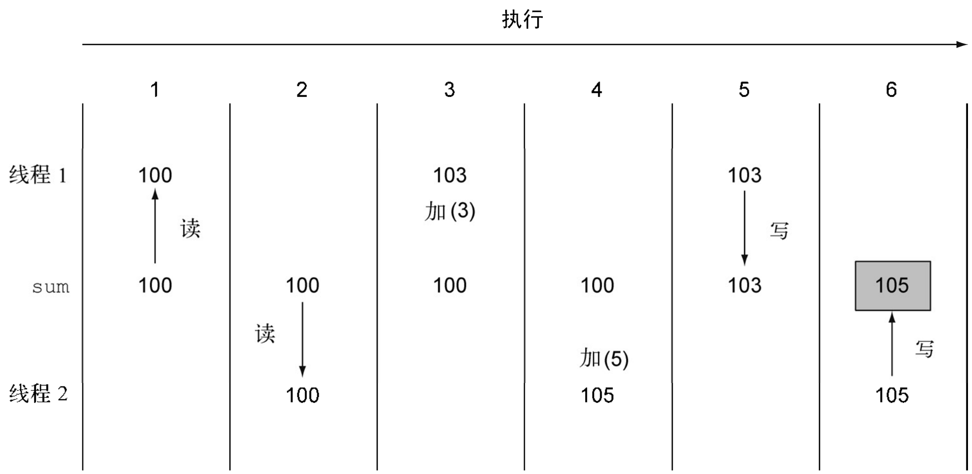
多线程并非易事

问题在于，通过多线程代码来利用并行（使用先前Java版本中的Thread API）并非易事。你得换一种思路：线程可能会同时访问并更新共享变量。因此，如果没有协调好[②]，数据可能会被意外改变。相比一步步执行的顺序模型，这个模型不太好理解[③]。比如，图1-5就展示了如果没有同步好，两个线程同时向共享变量sum加上一个数时，可能出现的问题。

① 从某种意义上说，这个名字不太好。一块多核芯片上的每个核都是一个五脏俱全的CPU。但“多核CPU”的说法很流行，所以我们就用*内核*来指代各个CPU。

② 传统上是利用synchronized关键字，但是要是用错了地方，就可能出现很多难以察觉的错误。Java 8基于Stream的并行提倡很少使用synchronized的函数式编程风格，它关注数据分块而不是协调访问。

③ 啊哈，促使语言发展的一个动力源！



线程1: `sum = sum + 3;`
线程2: `sum = sum + 5;`

图1-5　两个线程对共享的sum变量做加法的一种可能方式。结果是105，而不是预想的108

Java 8也用Stream API（java.util.stream）解决了这两个问题：集合处理时的套路和晦涩，以及难以利用多核。这样设计的第一个原因是，有许多反复出现的数据处理模式，类似于前一节所说的filterApples或SQL等数据库查询语言里熟悉的操作，如果在库中有这些就会很方便：根据标准筛选数据（比如较重的苹果），提取数据（例如抽取列表中每个苹果的重量字段），或给数据分组（例如，将一个数字列表分组，奇数和偶数分别列表）等。第二个原因是，这类操作常常可以并行化。例如，如图1-6所示，在两个CPU上筛选列表，可以让一个CPU处理列表的前一半，第二个CPU处理后一半，这称为分支步骤(1)。CPU随后对各自的半个列表做筛选(2)。最后(3)，一个CPU会把两个结果合并起来（Google搜索这么快就与此紧密相关，当然他们用的CPU远远不止两个了）。

到这里，我们只是说新的Stream API和Java现有的集合API的行为差不多：它们都能够访问数据项目的序列。不过，现在最好记得，Collection主要是为了存储和访问数据，而Stream则主要用于描述对数据的计算。这里的关键点在于，Stream允许并提倡并行处理一个Stream中的元素。虽然可能乍看上去有点儿怪，但筛选一个Collection（将上一节的filterApples应用在一个List上）的最快方法常常是将其转换为Stream，进行并行处理，然后再转换回List，下面举的串行和并行的例子都是如此。我们这里还只是说“几乎免费的并行”，让你稍微体验一下，如何利用Stream和Lambda表达式顺序或并行地从一个列表里筛选比较重的苹果。

顺序处理：

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
```

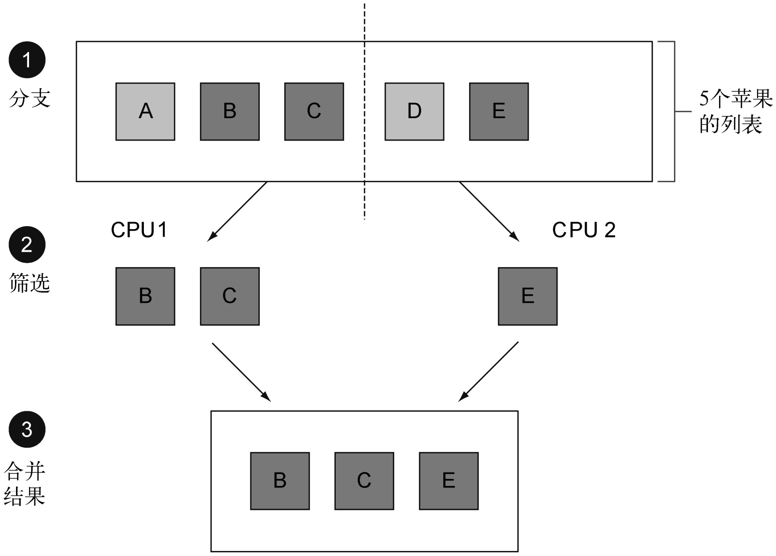


图1-6　将filter分支到两个CPU上并聚合结果

并行处理：

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

第7章会更详细地探讨Java 8中的并行数据处理及其特点。在加入所有这些新玩意儿改进Java的时候，Java 8设计者发现的一个现实问题就是现有的接口也在改进。比如，Collections.sort方法真的应该属于List接口，但却从来没有放在后者里。理想的情况下，你会希望做list.sort(comparator)，而不是Collections.sort(list, comparator)。这看起来无关紧要，但是在Java 8之前，你可能会更新一个接口，然后发现你把所有实现它的类也给更新了——简直是逻辑灾难！这个问题在Java 8里由默认方法解决了。

Java中的并行与无共享可变状态

大家都说Java里面并行很难，而且和synchronized相关的玩意儿都容易出问题。那Java 8里面有什么"灵丹妙药"呢？事实上有两个。首先，库会负责分块，即把大的流分成几个小的流，以便并行处理。其次，流提供的这个几乎免费的并行，只有在传递给filter之类的库方法的方法不会互动（比方说有可变的共享对象）时才能工作。但是其实这个限制对于程序员来说挺自然的，举个例子，我们的Apple::isGreenApple就是这样。确实，虽然**函数式编程**中的**函数**的主要意思是"把函数作为一等值"，不过它也常常隐含着第二层意思，即"执行时在元素之间无互动"。

1.4 默认方法

Java 8中加入默认方法主要是为了支持库设计师，让他们能够写出更容易改进的接口。这一点会在第9章中详谈。这一方法很重要，因为你会在接口中遇到越来越多的默认方法，但由于真正需要编写默认方法的程序员相对较少，而且它们只是有助于程序改进，而不是用于编写任何具体的程序，我们这里还是不要啰嗦了，举个例子吧。

在1.3节中，我们给出了下面这段Java 8示例代码：

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

但这里有个问题：在Java 8之前，`List<T>`并没有`stream`或`parallelStream`方法，它实现的`Collection<T>`接口也没有，因为当初还没有想到这些方法嘛！可没有这些方法，这些代码就不能编译。换作你自己的接口的话，最简单的解决方案就是让Java 8的设计者把`stream`方法加入`Collection`接口，并加入`ArrayList`类的实现。

可要是这样做，对用户来说就是噩梦了。有很多的替代集合框架都用Collection API实现了接口。但给接口加入一个新方法，意味着所有的实体类都必须为其提供一个实现。语言设计者没法控制`Collections`所有现有的实现，这下你就进退两难了：你如何改变已发布的接口而不破坏已有的实现呢？

Java 8的解决方法就是打破最后一环——接口如今可以包含实现类没有提供实现的方法签名了！那谁来实现它呢？缺失的方法主体随接口提供了（因此就有了默认实现），而不是由实现类提供。

这就给接口设计者提供了一个扩充接口的方式，而不会破坏现有的代码。Java 8在接口声明中使用新的`default`关键字来表示这一点。

例如，在Java 8里，你现在可以直接对`List`调用`sort`方法。它是用Java 8 `List`接口中如下所示的默认方法实现的，它会调用`Collections.sort`静态方法：

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

这意味着`List`的任何实体类都不需要显式实现`sort`，而在以前的Java版本中，除非提供了`sort`的实现，否则这些实体类在重新编译时都会失败。

不过慢着，一个类可以实现多个接口，不是吗？那么，如果在好几个接口里有多个默认实现，是否意味着Java中有了某种形式的多重继承？是的，在某种程度上是这样。我们在第9章中会谈到，Java 8用一些限制来避免出现类似于C++中臭名昭著的菱形继承问题。

1.5　来自函数式编程的其他好思想

前几节介绍了Java中从函数式编程中引入的两个核心思想：将方法和Lambda作为一等值，以及在没有可变共享状态时，函数或方法可以有效、安全地并行执行。前面说到的新的Stream API把这两种思想都用到了。

常见的函数式语言，如SML、OCaml、Haskell，还提供了进一步的结构来帮助程序员。其中之一就是通过使用更多的描述性数据类型来避免null。确实，计算机科学巨擘之一托尼·霍尔（Tony Hoare）在2009年伦敦QCon上的一个演讲中说道：

> 我把它叫作我的“价值亿万美金的错误”。就是在1965年发明了空引用……我无法抗拒放进一个空引用的诱惑，仅仅是因为它实现起来非常容易。

在Java 8里有一个Optional<T>类，如果你能一致地使用它的话，就可以帮助你避免出现NullPointer异常。它是一个容器对象，可以包含，也可以不包含一个值。Optional<T>中有方法来明确处理值不存在的情况，这样就可以避免NullPointer异常了。换句话说，它使用类型系统，允许你表明我们知道一个变量可能会没有值。我们会在第10章中详细讨论Optional<T>。

第二个想法是（结构）模式匹配[1]。这在数学中也有使用，例如：

```
f(0) = 1
f(n) = n*f(n-1) otherwise
```

在Java中，你可以在这里写一个if-then-else语句或一个switch语句。其他语言表明，对于更复杂的数据类型，模式匹配可以比if-then-else更简明地表达编程思想。对于这种数据类型，你也可以使用多态和方法重载来替代if-then-else，但对于哪种方式更合适，就语言设计而言仍有一些争论。[2]我们认为两者都是有用的工具，你都应该掌握。不幸的是，Java 8对模式匹配的支持并不完全，虽然我们会在第14章中介绍如何对其进行表达。与此同时，我们会用一个以Scala语言（另一个使用JVM的类Java语言，启发了Java在一些方面的发展；请参阅第15章）表达的例子加以描述。比方说，你要写一个程序对描述算术表达式的树做基本的简化。给定一个数据类型Expr代表这样的表达式，在Scala里你可以写以下代码，把Expr分解给它的各个部分，然后返回另一个Expr：

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    case BinOp("*", e, Number(1)) => e
    case BinOp("/", e, Number(1)) => e
    case _ => expr
}
```

加上0
乘以1
除以1
不能简化expr

① 这个术语有两个意思，这里我们指的是数学和函数式编程上所用的，即函数是分情况定义的，而不是使用if-then-else。它的另一个意思类似于“在给定目录中找到所有类似于IMG*.JPG形式的文件”，和所谓的正则表达式有关。

② 维基百科中文章“Expression Problem”（由Phil Wadler发明的术语）对这一讨论有所介绍。

这里，Scala的语法`expr match`就对应于Java中的`switch (expr)`。现在你不用担心这段代码，你可以在第14章阅读更多有关模式匹配的内容。现在，你可以把模式匹配看作`switch`的扩展形式，可以同时将一个数据类型分解成元素。

为什么Java中的`switch`语句应该限于原始类型值和`Strings`呢？函数式语言倾向于允许`switch`用在更多的数据类型上，包括允许模式匹配（在Scala代码中是通过`match`操作实现的）。在面向对象设计中，常用的访客模式可以用来遍历一组类（如汽车的不同组件：车轮、发动机、底盘等），并对每个访问的对象执行操作。模式匹配的一个优点是编译器可以报告常见错误，如："`Brakes`类属于用来表示`Car`类的组件的一族类。你忘记了要显式处理它。"

第13章和第14章给出了完整的教程，介绍函数式编程，以及如何在Java 8中编写函数式风格的程序，包括其库中提供的函数工具。第15章讨论Java 8的功能并与Scala进行比较。Scala和Java一样是在JVM上实现的，且近年来发展迅速，在编程语言生态系统中已经在一些方面威胁到了Java。这部分内容在书的后面几章，会让你进一步了解Java 8为什么加上了这些新功能。

1.6　小结

以下是你应从本章中学到的关键概念。

- 请记住语言生态系统的思想，以及语言面临的"要么改变，要么衰亡"的压力。虽然Java可能现在非常有活力，但你可以回忆一下其他曾经也有活力但未能及时改进的语言的命运，如COBOL。
- Java 8中新增的核心内容提供了令人激动的新概念和功能，方便我们编写既有效又简洁的程序。
- 现有的Java编程实践并不能很好地利用多核处理器。
- 函数是一等值；记得方法如何作为函数式值来传递，还有Lambda是怎样写的。
- Java 8中`Streams`的概念使得`Collections`的许多方面得以推广，让代码更为易读，并允许并行处理流元素。
- 你可以在接口中使用默认方法，在实现类没有实现方法时提供方法内容。
- 其他来自函数式编程的有趣思想，包括处理`null`和使用模式匹配。

第 2 章

通过行为参数化传递代码

本章内容

- 应对不断变化的需求
- 行为参数化
- 匿名类
- Lambda表达式预览
- 真实示例：Comparator、Runnable和GUI

在软件工程中，一个众所周知的问题就是，不管你做什么，用户的需求肯定会变。比方说，有个应用程序是帮助农民了解自己的库存的。这位农民可能想有一个查找库存中所有绿色苹果的功能。但到了第二天，他可能会告诉你："其实我还想找出所有重量超过150克的苹果。"又过了两天，农民又跑回来补充道："要是我可以找出所有既是绿色，重量也超过150克的苹果，那就太棒了。"你要如何应对这样不断变化的需求？理想的状态下，应该把你的工作量降到最少。此外，类似的新功能实现起来还应该很简单，而且易于长期维护。

*行为参数化*就是可以帮助你处理频繁变更的需求的一种软件开发模式。一言以蔽之，它意味着拿出一个代码块，把它准备好却不去执行它。这个代码块以后可以被你程序的其他部分调用，这意味着你可以推迟这块代码的执行。例如，你可以将代码块作为参数传递给另一个方法，稍后再去执行它。这样，这个方法的行为就基于那块代码被参数化了。例如，如果你要处理一个集合，可能会写一个方法：

- 可以对列表中的每个元素做"某件事"
- 可以在列表处理完后做"另一件事"
- 遇到错误时可以做"另外一件事"

*行为参数化*说的就是这个。打个比方吧：你的室友知道怎么开车去超市，再开回家。于是你可以告诉他去买一些东西，比如面包、奶酪、葡萄酒什么的。这相当于调用一个goAndBuy方法，把购物单作为参数。然而，有一天你在上班，你需要他去做一件他从来没有做过的事情：从邮局取一个包裹。现在你就需要传递给他一系列指示了：去邮局，使用单号，和工作人员说明情况，取走包裹。你可以把这些指示用电子邮件发给他，当他收到之后就可以按照指示行事了。你现在做的事情就更高级一些了，相当于一个方法：go，它可以接受不同的新行为作为参数，然后去执行。

这一章首先会给你讲解一个例子，说明如何对你的代码加以改进，从而更灵活地适应不断变化的需求。在此基础之上，我们将展示如何把行为参数化用在几个真实的例子上。比如，你可能已经用过了行为参数化模式——使用Java API中现有的类和接口，对`List`进行排序，筛选文件名，或告诉一个`Thread`去执行代码块，甚或是处理GUI事件。你很快会发现，在Java中使用这种模式十分啰嗦。Java 8中的Lambda解决了代码啰嗦的问题。我们会在第3章中向你展示如何构建Lambda表达式、其使用场合，以及如何利用它让代码更简洁。

2.1　应对不断变化的需求

编写能够应对变化的需求的代码并不容易。让我们来看一个例子，我们会逐步改进这个例子，以展示一些让代码更灵活的最佳做法。就农场库存程序而言，你必须实现一个从列表中筛选绿苹果的功能。听起来很简单吧？

2.1.1　初试牛刀：筛选绿苹果

第一个解决方案可能是下面这样的：

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<Apple>();   <—— 累积苹果的列表
    for(Apple apple: inventory){
        if( "green".equals(apple.getColor() ) {   <—— 仅仅选出绿苹果
            result.add(apple);
        }
    }
    return result;
}
```

突出显示的行就是筛选绿苹果所需的条件。但是现在农民改主意了，他还想要筛选红苹果。你该怎么做呢？简单的解决办法就是复制这个方法，把名字改成`filterRedApples`，然后更改`if`条件来匹配红苹果。然而，要是农民想要筛选多种颜色：浅绿色、暗红色、黄色等，这种方法就应付不了了。一个良好的原则是在编写类似的代码之后，尝试将其抽象化。

2.1.2　再展身手：把颜色作为参数

一种做法是给方法加一个参数，把颜色变成参数，这样就能灵活地适应变化了：

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
                                              String color) {
    List<Apple> result = new ArrayList<Apple>();
    for (Apple apple: inventory){
        if ( apple.getColor().equals(color) ) {
            result.add(apple);
        }
    }
    return result;
}
```

现在，只要像下面这样调用方法，农民朋友就会满意了：

```
List<Apple> greenApples = filterApplesByColor(inventory, "green");
List<Apple> redApples = filterApplesByColor(inventory, "red");
…
```

太简单了对吧？让我们把例子再弄得复杂一点儿。这位农民又跑回来和你说："要是能区分轻的苹果和重的苹果就太好了。重的苹果一般是重量大于150克。"

作为软件工程师，你早就想到农民可能会要改变重量，于是你写了下面的方法，用另一个参数来应对不同的重量：

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                               int weight) {
    List<Apple> result = new ArrayList<Apple>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ){
            result.add(apple);
        }
    }
    return result;
}
```

解决方案不错，但是请注意，你复制了大部分的代码来实现遍历库存，并对每个苹果应用筛选条件。这有点儿令人失望，因为它打破了DRY（Don't Repeat Yourself，不要重复自己）的软件工程原则。如果你想要改变筛选遍历方式来提升性能呢？那就得修改所有方法的实现，而不是只改一个。从工程工作量的角度来看，这代价太大了。

你可以将颜色和重量结合为一个方法，称为`filter`。不过就算这样，你还是需要一种方式来区分想要筛选哪个属性。你可以加上一个标志来区分对颜色和重量的查询（但绝不要这样做！我们很快会解释为什么）。

2.1.3　第三次尝试：对你能想到的每个属性做筛选

一种把所有属性结合起来的笨拙尝试如下所示：

```
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<Apple>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
             (!flag && apple.getWeight() > weight) ){      <—— 十分笨拙的选择颜色或重量的方式
            result.add(apple);
        }
    }
    return result;
}
```

你可以这么用（但真的很笨拙）：

```
List<Apple> greenApples = filterApples(inventory, "green", 0, true);
```

```
List<Apple> heavyApples = filterApples(inventory, "", 150, false);
…
```

这个解决方案再差不过了。首先，客户端代码看上去糟透了。true和false是什么意思？此外，这个解决方案还是不能很好地应对变化的需求。如果这位农民要求你对苹果的不同属性做筛选，比如大小、形状、产地等，又怎么办？而且，如果农民要求你组合属性，做更复杂的查询，比如绿色的重苹果，又该怎么办？你会有好多个重复的filter方法，或一个巨大的非常复杂的方法。到目前为止，你已经给filterApples方法加上了值（比如String、Integer或boolean）的参数。这对于某些确定性问题可能还不错。但如今这种情况下，你需要一种更好的方式，来把苹果的选择标准告诉你的filterApples方法。在下一节中，我们会介绍了如何利用行为参数化实现这种灵活性。

2.2　行为参数化

你在上一节中已经看到了，你需要一种比添加很多参数更好的方法来应对变化的需求。让我们后退一步来看看更高层次的抽象。一种可能的解决方案是对你的选择标准建模：你考虑的是苹果，需要根据Apple的某些属性（比如它是绿色的吗？重量超过150克吗？）来返回一个boolean值。我们把它称为谓词（即一个返回boolean值的函数）。让我们定义一个接口来对选择标准建模：

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

现在你就可以用ApplePredicate的多个实现代表不同的选择标准了，比如（如图2-1所示）：

```
public class AppleHeavyWeightPredicate implements ApplePredicate{   <-- 仅仅选出重的苹果
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate{   <-- 仅仅选出绿苹果
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
```

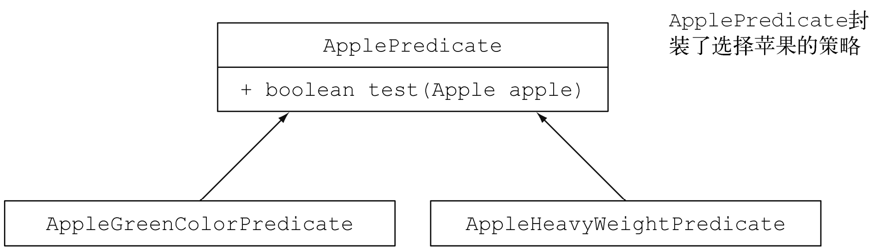


图2-1　选择苹果的不同策略

你可以把这些标准看作filter方法的不同行为。你刚做的这些和“策略设计模式”[1]相关，它让你定义一族算法，把它们封装起来（称为“策略”），然后在运行时选择一个算法。在这里，算法族就是ApplePredicate，不同的策略就是AppleHeavyWeightPredicate和AppleGreen-ColorPredicate。

但是，该怎么利用ApplePredicate的不同实现呢？你需要filterApples方法接受ApplePredicate对象，对Apple做条件测试。这就是行为参数化：让方法接受多种行为（或战略）作为参数，并在内部使用，来完成不同的行为。

要在我们的例子中实现这一点，你要给filterApples方法添加一个参数，让它接受ApplePredicate对象。这在软件工程上有很大好处：现在你把filterApples方法迭代集合的逻辑与你要应用到集合中每个元素的行为（这里是一个谓词）区分开了。

第四次尝试：根据抽象条件筛选

利用ApplePredicate改过之后，filter方法看起来是这样的：

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p){
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if(p.test(apple)){          <——
            result.add(apple);
        }
    }
    return result;
}
```

谓词对象封装了测试苹果的条件

1. 传递代码/行为

这里值得停下来小小地庆祝一下。这段代码比我们第一次尝试的时候灵活多了，读起来、用起来也更容易！现在你可以创建不同的ApplePredicate对象，并将它们传递给filterApples方法。免费的灵活性！比如，如果农民让你找出所有重量超过150克的红苹果，你只需要创建一个类来实现ApplePredicate就行了。你的代码现在足够灵活，可以应对任何涉及苹果属性的需求变更了：

```
public class AppleRedAndHeavyPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return "red".equals(apple.getColor())
            && apple.getWeight() > 150;
    }
}

List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

你已经做成了一件很酷的事：filterApples方法的行为取决于你通过ApplePredicate对

① 见http://en.wikipedia.org/wiki/Strategy_pattern。

象传递的代码。换句话说，你把filterApples方法的行为参数化了！

请注意，在上一个例子中，唯一重要的代码是test方法的实现，如图2-2所示；正是它定义了filterApples方法的新行为。但令人遗憾的是，由于该filterApples方法只能接受对象，所以你必须把代码包裹在ApplePredicate对象里。你的做法就类似于在内联“传递代码”，因为你是通过一个实现了test方法的对象来传递布尔表达式的。你将在2.3节（第3章中有更详细的内容）中看到，通过使用Lambda，你可以直接把表达式"red".equals(apple.getColor()) &&apple.getWeight() > 150传递给filterApples方法，而无需定义多个ApplePredicate类，从而去掉不必要的代码。

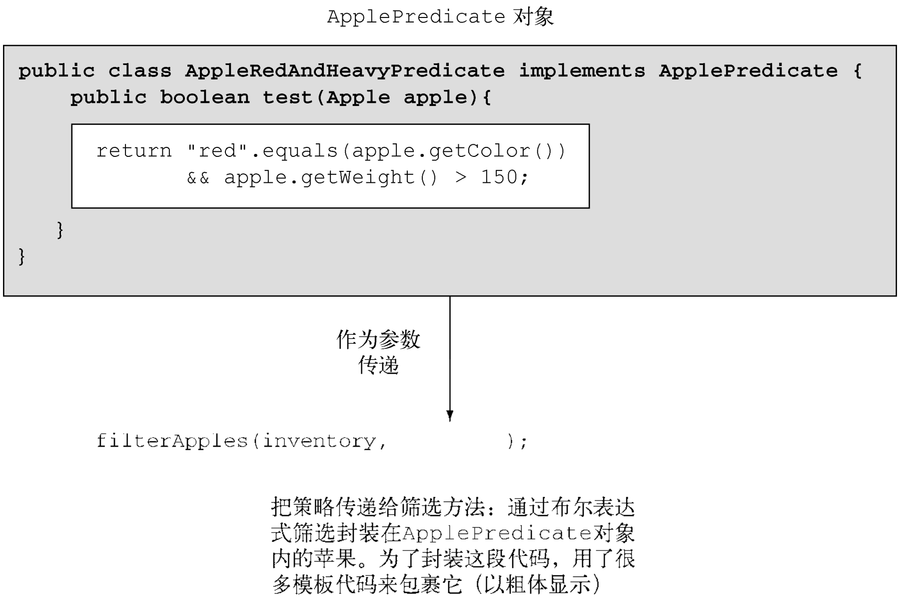


图2-2 参数化filterApples的行为，并传递不同的筛选策略

2. 多种行为，一个参数

正如我们先前解释的那样，行为参数化的好处在于你可以把迭代要筛选的集合的逻辑与对集合中每个元素应用的行为区分开来。这样你可以重复使用同一个方法，给它不同的行为来达到不同的目的，如图2-3所示。

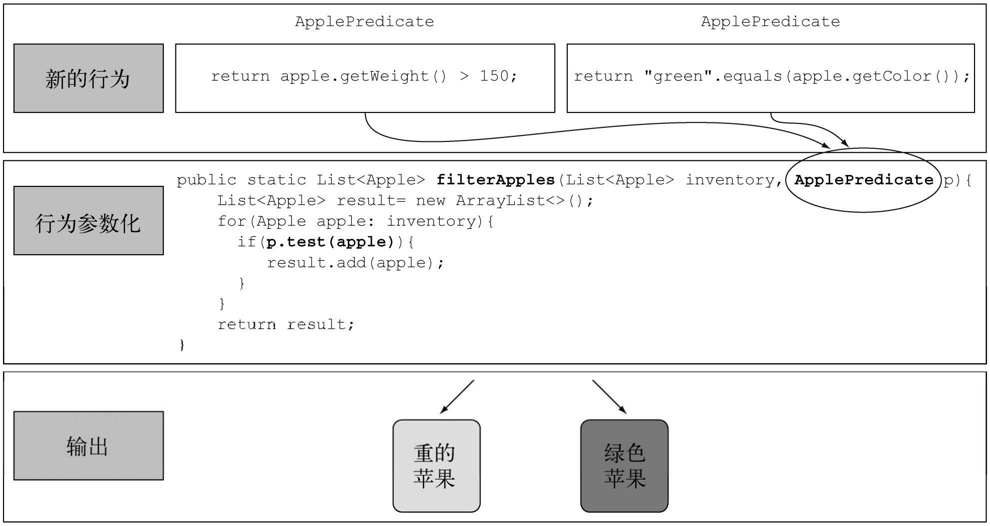


图2-3　参数化`filterApples`的行为并传递不同的筛选策略

这就是说行为参数化是一个有用的概念的原因。你应该把它放进你的工具箱里，用来编写灵活的API。

为了保证你对行为参数化运用自如，看看测验2.1吧！

测验2.1：编写灵活的`prettyPrintApple`方法

编写一个`prettyPrintApple`方法，它接受一个`Apple`的`List`，并可以对它参数化，以多种方式根据苹果生成一个`String`输出（有点儿像多个可定制的`toString`方法）。例如，你可以告诉`prettyPrintApple`方法，只打印每个苹果的重量。此外，你可以让`prettyPrintApple`方法分别打印每个苹果，然后说明它是重的还是轻的。解决方案和我们前面讨论的筛选的例子类似。为了帮你上手，我们提供了`prettyPrintApple`方法的一个粗略的框架：

```
public static void prettyPrintApple(List<Apple> inventory, ???){
    for(Apple apple: inventory) {
        String output = ???.???(apple);
        System.out.println(output);
    }
}
```

答案如下。

首先，你需要一种表示接受`Apple`并返回一个格式`String`值的方法。前面我们在编写`ApplePredicate`接口的时候，写过类似的东西：

```
public interface AppleFormatter{
    String accept(Apple a);
}
```

现在你就可以通过实现`AppleFormatter`方法，来表示多种格式行为了：

```
public class AppleFancyFormatter implements AppleFormatter{
    public String accept(Apple apple){
        String characteristic = apple.getWeight() > 150 ? "heavy" :
            "light";
        return "A " + characteristic +
               " " + apple.getColor() +" apple";
    }
}
public class AppleSimpleFormatter implements AppleFormatter{
    public String accept(Apple apple){
        return "An apple of " + apple.getWeight() + "g";
    }
}
```

最后，你需要告诉prettyPrintApple方法接受AppleFormatter对象，并在内部使用它们。你可以给prettyPrintApple加上一个参数：

```
public static void prettyPrintApple(List<Apple> inventory,
                                    AppleFormatter formatter){
    for(Apple apple: inventory){
        String output = formatter.accept(apple);
        System.out.println(output);
    }
}
```

搞定啦！现在你就可以给prettyPrintApple方法传递多种行为了。为此，你首先要实例化AppleFormatter的实现，然后把它们作为参数传给prettyPrintApple：

```
prettyPrintApple(inventory, new AppleFancyFormatter());
```

这将产生一个类似于下面的输出：

```
A light green apple
A heavy red apple
…
```

或者试试这个：

```
prettyPrintApple(inventory, new AppleSimpleFormatter());
```

这将产生一个类似于下面的输出：

```
An apple of 80g
An apple of 155g
…
```

你已经看到，可以把行为抽象出来，让你的代码适应需求的变化，但这个过程很啰嗦，因为你需要声明很多只要实例化一次的类。让我们来看看可以怎样改进。

2.3 对付啰嗦

我们都知道，人们都不愿意用那些很麻烦的功能或概念。目前，当要把新的行为传递给filterApples方法的时候，你不得不声明好几个实现ApplePredicate接口的类，然后实例化好几个只会提到一次的ApplePredicate对象。下面的程序总结了你目前看到的一切。这真是很啰嗦，很费时间！

代码清单2-1　行为参数化：用谓词筛选苹果

```
public class AppleHeavyWeightPredicate implements ApplePredicate{   <-- 选择较重苹果的谓词
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate{   <-- 选择绿苹果的谓词
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
public class FilteringApples{
    public static void main(String...args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));
        List<Apple> heavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());   <-- 结果是一个包含一个155克Apple的List
        List<Apple> greenApples =
            filterApples(inventory, new AppleGreenColorPredicate());   <-- 结果是一个包含两个绿Apple的List
    }
    public static List<Apple> filterApples(List<Apple> inventory,
                                           ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for (Apple apple : inventory){
            if (p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

费这么大劲儿真没必要，能不能做得更好呢？Java有一个机制称为*匿名类*，它可以让你同时声明和实例化一个类。它可以帮助你进一步改善代码，让它变得更简洁。但这也不完全令人满意。2.3.3节简短地介绍了Lambda表达式如何让你的代码更易读，我们将在下一章详细讨论。

2.3.1　匿名类

*匿名类*和你熟悉的Java局部类（块中定义的类）差不多，但匿名类没有名字。它允许你同时声明并实例化一个类。换句话说，它允许你随用随建。

2.3.2　第五次尝试：使用匿名类

下面的代码展示了如何通过创建一个用匿名类实现ApplePredicate的对象，重写筛选的例子：

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {   <-- 直接内联参数化filterapples方法的行为
    public boolean test(Apple apple){
        return "red".equals(apple.getColor());
    }
});
```

GUI应用程序中经常使用匿名类来创建事件处理器对象（下面的例子使用的是Java FX API，一种现代的Java UI平台）：

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Woooo a click!!");
    }
});
```

但匿名类还是不够好。第一，它往往很笨重，因为它占用了很多空间。还拿前面的例子来看，如下面高亮的代码所示：

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {   <—
    public boolean test(Apple a){
        return "red".equals(a.getColor());                               很多模板
    }                                                                    代码
});
button.setOnAction(new EventHandler<ActionEvent>() {                      <—
    public void handle(ActionEvent event) {
        System.out.println("Woooo a click!!");
    }
});
```

第二，很多程序员觉得它用起来很让人费解。比如，测验2.2展示了一个经典的Java谜题，它让大多数程序员都措手不及。你来试试看吧。

测验2.2：匿名类谜题

下面的代码执行时会有什么样的输出呢，4、5、6还是42？

```
public class MeaningOfThis
{
    public final int value = 4;
    public void doIt()
    {
        int value = 6;
        Runnable r = new Runnable(){
                public final int value = 5;
                public void run(){
                    int value = 10;
                    System.out.println(this.value);
                }
            };
            r.run();
        }
        public static void main(String...args)
        {
            MeaningOfThis m = new MeaningOfThis();      这一行的输
            m.doIt();                               <—  出是什么？
        }
}
```

答案是5，因为this指的是包含它的Runnable，而不是外面的类MeaningOfThis。

整体来说，啰嗦就不好；它让人不愿意使用语言的某种功能，因为编写和维护啰嗦的代码需要很长时间，而且代码也不易读。好的代码应该是一目了然的。即使匿名类处理在某种程度上改善了为一个接口声明好几个实体类的啰嗦问题，但它仍不能令人满意。在只需要传递一段简单的代码时（例如表示选择标准的boolean表达式），你还是要创建一个对象，明确地实现一个方法来定义一个新的行为（例如Predicate中的test方法或是EventHandler中的handler方法）。

在理想的情况下，我们想鼓励程序员使用行为参数化模式，因为正如你在前面看到的，它让代码更能适应需求的变化。在第3章中，你会看到Java 8的语言设计者通过引入Lambda表达式——一种更简洁的传递代码的方式——解决了这个问题。好了，悬念够多了，下面简单介绍一下Lambda表达式是怎么让代码更干净的。

2.3.3　第六次尝试：使用 Lambda 表达式

上面的代码在Java 8里可以用Lambda表达式重写为下面的样子：

```
List<Apple> result =
    filterApples(inventory, (Apple apple) -> "red".equals(apple.getColor()));
```

不得不承认这代码看上去比先前干净很多。这很好，因为它看起来更像问题陈述本身了。我们现在已经解决了啰嗦的问题。图2-4对我们到目前为止的工作做了一个小结。

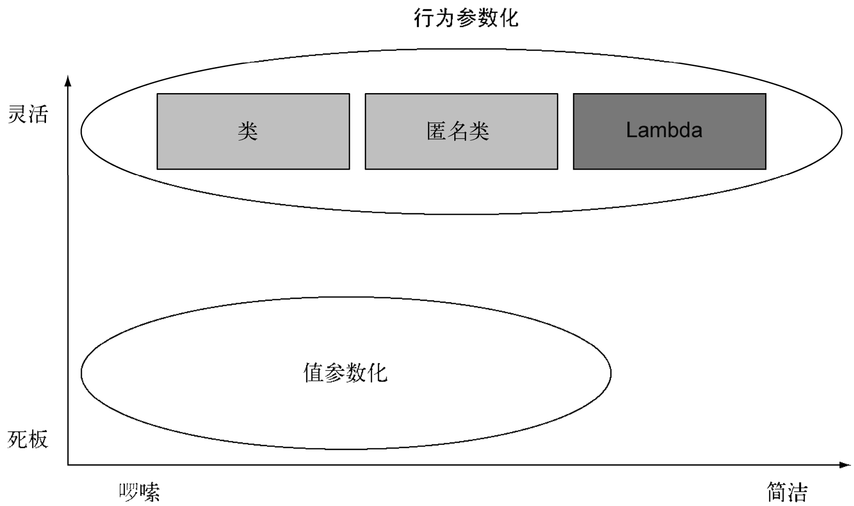


图2-4　行为参数化与值参数化

2.3.4 第七次尝试：将 List 类型抽象化

在通往抽象的路上，我们还可以更进一步。目前，filterApples方法还只适用于Apple。你还可以将List类型抽象化，从而超越你眼前要处理的问题：

```
public interface Predicate<T>{
    boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p){   ←—— 引入类型参数T
    List<T> result = new ArrayList<>();
    for(T e: list){
        if(p.test(e)){
            result.add(e);
        }
    }
    return result;
}
```

现在你可以把filter方法用在香蕉、桔子、Integer或是String的列表上了。这里有一个使用Lambda表达式的例子：

```
List<Apple> redApples =
    filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<Integer> evenNumbers =
    filter(numbers, (Integer i) -> i % 2 == 0);
```

酷不酷？你现在在灵活性和简洁性之间找到了最佳平衡点，这在Java 8之前是不可能做到的！

2.4 真实的例子

你现在已经看到，行为参数化是一个很有用的模式，它能够轻松地适应不断变化的需求。这种模式可以把一个行为（一段代码）封装起来，并通过传递和使用创建的行为（例如对Apple的不同谓词）将方法的行为参数化。前面提到过，这种做法类似于策略设计模式。你可能已经在实践中用过这个模式了。Java API中的很多方法都可以用不同的行为来参数化。这些方法往往与匿名类一起使用。我们会展示三个例子，这应该能帮助你巩固传递代码的思想了：用一个Comparator排序，用Runnable执行一个代码块，以及GUI事件处理。

2.4.1 用 Comparator 来排序

对集合进行排序是一个常见的编程任务。比如，你的那位农民朋友想要根据苹果的重量对库存进行排序，或者他可能改了主意，希望你根据颜色对苹果进行排序。听起来有点儿耳熟？是的，你需要一种方法来表示和使用不同的排序行为，来轻松地适应变化的需求。

在Java 8中，List自带了一个sort方法（你也可以使用Collections.sort）。sort的行为可以用java.util.Comparator对象来参数化，它的接口如下：

```
// java.util.Comparator
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

因此，你可以随时创建Comparator的实现，用sort方法表现出不同的行为。比如，你可以使用匿名类，按照重量升序对库存排序：

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

如果农民改了主意，你可以随时创建一个Comparator来满足他的新要求，并把它传递给sort方法。而如何进行排序这一内部细节都被抽象掉了。用Lambda表达式的话，看起来就是这样：

```
inventory.sort(
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

现在暂时不用担心这个新语法，下一章我们会详细讲解如何编写和使用Lambda表达式。

2.4.2　用 Runnable 执行代码块

线程就像是轻量级的进程：它们自己执行一个代码块。但是，怎么才能告诉线程要执行哪块代码呢？多个线程可能会运行不同的代码。我们需要一种方式来代表稍候执行的一段代码。在Java里，你可以使用Runnable接口表示一个要执行的代码块。请注意，代码不会返回任何结果（即void）：

```
// java.lang.Runnable
public interface Runnable{
    public void run();
}
```

你可以像下面这样，使用这个接口创建执行不同行为的线程：

```
Thread t = new Thread(new Runnable() {
    public void run(){
        System.out.println("Hello world");
    }
});
```

用Lambda表达式的话，看起来是这样：

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

2.4.3　GUI 事件处理

GUI编程的一个典型模式就是执行一个操作来响应特定事件，如鼠标单击或在文字上悬停。

例如，如果用户单击“发送”按钮，你可能想显示一个弹出式窗口，或把行为记录在一个文件中。你还是需要一种方法来应对变化；你应该能够作出任意形式的响应。在JavaFX中，你可以使用EventHandler，把它传给setOnAction来表示对事件的响应：

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

这里，setOnAction方法的行为就用EventHandler参数化了。用Lambda表达式的话，看起来就是这样：

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

2.5　小结

以下是你应从本章中学到的关键概念。

- 行为参数化，就是一个方法接受多个不同的行为作为参数，并在内部使用它们，完成不同行为的能力。
- 行为参数化可让代码更好地适应不断变化的要求，减轻未来的工作量。
- 传递代码，就是将新行为作为参数传递给方法。但在Java 8之前这实现起来很啰嗦。为接口声明许多只用一次的实体类而造成的啰嗦代码，在Java 8之前可以用匿名类来减少。
- Java API包含很多可以用不同行为进行参数化的方法，包括排序、线程和GUI处理。

第 3 章 Lambda表达式

本章内容

- ❑ Lambda管中窥豹
- ❑ 在哪里以及如何使用Lambda
- ❑ 环绕执行模式
- ❑ 函数式接口，类型推断
- ❑ 方法引用
- ❑ Lambda复合

在上一章中，你了解了利用行为参数化来传递代码有助于应对不断变化的需求。它允许你定义一个代码块来表示一个行为，然后传递它。你可以决定在某一事件发生时（例如单击一个按钮）或在算法中的某个特定时刻（例如筛选算法中类似于“重量超过150克的苹果”的谓词，或排序中的自定义比较操作）运行该代码块。一般来说，利用这个概念，你就可以编写更为灵活且可重复使用的代码了。

但你也看到，使用匿名类来表示不同的行为并不令人满意：代码十分啰嗦，这会影响程序员在实践中使用行为参数化的积极性。在本章中，我们会教给你Java 8中解决这个问题的新工具——Lambda表达式。它可以让你很简洁地表示一个行为或传递代码。现在你可以把Lambda表达式看作匿名功能，它基本上就是没有声明名称的方法，但和匿名类一样，它也可以作为参数传递给一个方法。

我们会展示如何构建Lambda，它的使用场合，以及如何利用它使代码更简洁。我们还会介绍一些新的东西，如类型推断和Java 8 API中重要的新接口。最后，我们将介绍方法引用（method reference），这是一个常常和Lambda表达式联用的有用的新功能。

本章的行文思想就是教你如何一步一步地写出更简洁、更灵活的代码。在本章结束时，我们会把所有教过的概念融合在一个具体的例子里：我们会用Lambda表达式和方法引用逐步改进第2章中的排序例子，使之更加简明易读。这一章很重要，而且你将在本书中大量使用Lambda。

3.1 Lambda 管中窥豹

可以把Lambda表达式理解为简洁地表示可传递的匿名函数的一种方式：它没有名称，但它有参数列表、函数主体、返回类型，可能还有一个可以抛出的异常列表。这个定义够大的，让我们慢慢道来。

- 匿名——我们说匿名，是因为它不像普通的方法那样有一个明确的名称：写得少而想得多！
- 函数——我们说它是函数，是因为Lambda函数不像方法那样属于某个特定的类。但和方法一样，Lambda有参数列表、函数主体、返回类型，还可能有可以抛出的异常列表。
- 传递——Lambda表达式可以作为参数传递给方法或存储在变量中。
- 简洁——无需像匿名类那样写很多模板代码。

你是不是好奇Lambda这个词是从哪儿来的？其实它来自于学术界开发出来的一套用来描述计算的λ演算法。你为什么应该关心Lambda表达式呢？你在上一章中看到了，在Java中传递代码十分繁琐和冗长。那么，现在有了好消息！Lambda解决了这个问题：它可以让你十分简明地传递代码。理论上来说，你在Java 8之前做不了的事情，Lambda也做不了。但是，现在你用不着再用匿名类写一堆笨重的代码，来体验行为参数化的好处了！Lambda表达式鼓励你采用我们上一章中提到的行为参数化风格。最终结果就是你的代码变得更清晰、更灵活。比如，利用Lambda表达式，你可以更为简洁地自定义一个Comparator对象。

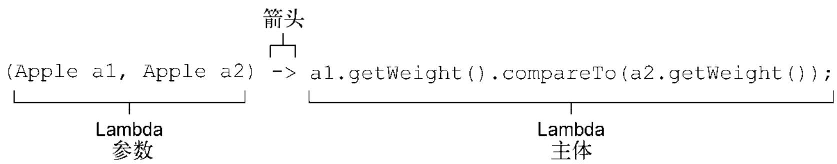


图3-1 Lambda表达式由参数、箭头和主体组成

先前：

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

之后（用了Lambda表达式）：

```
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

不得不承认，代码看起来更清晰了！要是现在你觉得Lambda表达式看起来一头雾水的话也没关系，我们很快会一点点解释清楚的。现在，请注意你基本上只传递了比较两个苹果重量所真正需要的代码。看起来就像是只传递了compare方法的主体。你很快就会学到，你甚至还可以进

一步简化代码。我们将在下一节解释在哪里以及如何使用Lambda表达式。

我们刚刚展示给你的Lambda表达式有三个部分，如图3-1所示。

- 参数列表——这里它采用了Comparator中compare方法的参数，两个Apple。
- 箭头——箭头->把参数列表与Lambda主体分隔开。
- Lambda主体——比较两个Apple的重量。表达式就是Lambda的返回值了。

为了进一步说明，下面给出了Java 8中五个有效的Lambda表达式的例子。

代码清单3-1　Java 8中有效的Lambda表达式

```
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x+y);
}

() -> 42
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

第一个Lambda表达式具有一个String类型的参数并返回一个int。Lambda没有return语句，因为已经隐含了return

第二个Lambda表达式有一个Apple类型的参数并返回一个boolean（苹果的重量是否超过150克）

第三个Lambda表达式具有两个int类型的参数而没有返回值（void返回）。注意Lambda表达式可以包含多行语句，这里是两行

第四个Lambda表达式没有参数，返回一个int

第五个Lambda表达式具有两个Apple类型的参数，返回一个int：比较两个Apple的重量

Java语言设计者选择这样的语法，是因为C#和Scala等语言中的类似功能广受欢迎。Lambda的基本语法是

```
(parameters) -> expression
```

或（请注意语句的花括号）

```
(parameters) -> { statements; }
```

你可以看到，Lambda表达式的语法很简单。做一下测验3.1，看看自己是不是理解了这个模式。

测验3.1：Lambda语法

根据上述语法规则，以下哪个不是有效的Lambda表达式？

(1) `() -> {}`

(2) `() -> "Raoul"`

(3) `() -> {return "Mario";}`

(4) `(Integer i) -> return "Alan" + i;`

(5) `(String s) -> {"IronMan";}`

答案：只有4和5是无效的Lambda。

(1) 这个Lambda没有参数，并返回void。它类似于主体为空的方法：public void run() {}。

(2) 这个Lambda没有参数，并返回String作为表达式。

(3) 这个Lambda没有参数，并返回String（利用显式返回语句）。

(4) `return`是一个控制流语句。要使此Lambda有效，需要使花括号，如下所示：`(Integer i) -> {return "Alan" + i;}`。

(5) “Iron Man”是一个表达式，不是一个语句。要使此Lambda有效，你可以去除花括号和分号，如下所示：`(String s) -> "Iron Man"`。或者如果你喜欢，可以使用显式返回语句，如下所示：`(String s)->{return "IronMan";}`。

表3-1提供了一些Lambda的例子和使用案例。

表3-1 Lambda示例

使用案例	Lambda示例
布尔表达式	`(List<String> list) -> list.isEmpty()`
创建对象	`() -> new Apple(10)`
消费一个对象	`(Apple a) -> { System.out.println(a.getWeight()); }`
从一个对象中选择/抽取	`(String s) -> s.length()`
组合两个值	`(int a, int b) -> a * b`
比较两个对象	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`

3.2 在哪里以及如何使用 Lambda

现在你可能在想，在哪里可以使用Lambda表达式。在上一个例子中，你把Lambda赋给了一个`Comparator<Apple>`类型的变量。你也可以在上一章中实现的`filter`方法中使用Lambda：

```
List<Apple> greenApples =
    filter(inventory, (Apple a) -> "green".equals(a.getColor()));
```

那到底在哪里可以使用Lambda呢？你可以在函数式接口上使用Lambda表达式。在上面的代码中，你可以把Lambda表达式作为第二个参数传给`filter`方法，因为它这里需要`Predicate<T>`，而这是一个函数式接口。如果这听起来太抽象，不要担心，现在我们就来详细解释这是什么意思，以及函数式接口是什么。

3.2.1 函数式接口

还记得你在第2章里，为了参数化`filter`方法的行为而创建的`Predicate<T>`接口吗？它就是一个函数式接口！为什么呢？因为`Predicate`仅仅定义了一个抽象方法：

```
public interface Predicate<T>{
    boolean test (T t);
}
```

一言以蔽之，函数式接口就是只定义一个抽象方法的接口。你已经知道了Java API中的一些其他函数式接口，如我们在第2章中谈到的Comparator和Runnable。

```
public interface Comparator<T> {        <—— java.util.Comparator
    int compare(T o1, T o2);
}

public interface Runnable{              <—— java.lang.Runnable
    void run();
}

public interface ActionListener extends EventListener{         <——
    void actionPerformed(ActionEvent e);     java.awt.event.ActionListener
}

public interface Callable<V>{       <—— java.util.concurrent.Callable
    V call();
}

public interface PrivilegedAction<V>{    <—— java.security.PrivilegedAction
    V run();
}
```

注意　你将会在第9章中看到，接口现在还可以拥有**默认方法**（即在类没有对方法进行实现时，其主体为方法提供默认实现的方法）。哪怕有很多默认方法，只要接口只定义了一个**抽象方法**，它就仍然是一个函数式接口。

为了检查你的理解程度，测验3.2将帮助你测试自己是否掌握了函数式接口的概念。

测验3.2：函数式接口

下面哪些接口是函数式接口？

```
public interface Adder{
    int add(int a, int b);
}
public interface SmartAdder extends Adder{
    int add(double a, double b);
}
public interface Nothing{
}
```

答案：只有Adder是函数式接口。

SmartAdder不是函数式接口，因为它定义了两个叫作add的抽象方法（其中一个是从Adder那里继承来的）。

Nothing也不是函数式接口，因为它没有声明抽象方法。

用函数式接口可以干什么呢？Lambda表达式允许你直接以内联的形式为函数式接口的抽象

方法提供实现，并把整个表达式作为函数式接口的实例（具体说来，是函数式接口一个具体实现的实例）。你用匿名内部类也可以完成同样的事情，只不过比较笨拙：需要提供一个实现，然后再直接内联将它实例化。下面的代码是有效的，因为Runnable是一个只定义了一个抽象方法run的函数式接口：

```
Runnable r1 = () -> System.out.println("Hello World 1");   <— 使用Lambda

Runnable r2 = new Runnable(){                              <— 使用匿名类
    public void run(){
        System.out.println("Hello World 2");
    }
};

public static void process(Runnable r){
    r.run();
}
process(r1);          <— 打印“Hello World 1”
process(r2);          <— 打印“Hello World 2”
process(() -> System.out.println("Hello World 3"));   <— 利用直接传递的Lambda打印“Hello World 3”
```

3

3.2.2　函数描述符

函数式接口的抽象方法的签名基本上就是Lambda表达式的签名。我们将这种抽象方法叫作*函数描述符*。例如，Runnable接口可以看作一个什么也不接受什么也不返回（void）的函数的签名，因为它只有一个叫作run的抽象方法，这个方法什么也不接受，什么也不返回（void）。[①]

我们在本章中使用了一个特殊表示法来描述Lambda和函数式接口的签名。() -> void代表了参数列表为空，且返回void的函数。这正是Runnable接口所代表的。举另一个例子，(Apple, Apple) -> int代表接受两个Apple作为参数且返回int的函数。我们会在3.4节和本章后面的表3-2中提供关于函数描述符的更多信息。

你可能已经在想，Lambda表达式是怎么做类型检查的。我们会在3.5节中详细介绍，编译器是如何检查Lambda在给定上下文中是否有效的。现在，只要知道Lambda表达式可以被赋给一个变量，或传递给一个接受函数式接口作为参数的方法就好了，当然这个Lambda表达式的签名要和函数式接口的抽象方法一样。比如，在我们之前的例子里，你可以像下面这样直接把一个Lambda传给process方法：

```
public void process(Runnable r){
    r.run();
}

process(() -> System.out.println("This is awesome!!"));
```

此代码执行时将打印“This is awesome!!”。Lambda表达式()-> System.out.println

① Scala等语言的类型系统提供显式类型标注，可以描述函数的类型（称为“函数类型”）。Java重用了函数式接口提供的标准类型，并将其映射成一种形式的函数类型。

("This is awesome!!")不接受参数且返回void。 这恰恰是Runnable接口中run方法的签名。

你可能会想:“为什么只有在需要函数式接口的时候才可以传递Lambda呢?”语言的设计者也考虑过其他办法,例如给Java添加函数类型(有点儿像我们介绍的描述Lambda表达式签名的特殊表示法,我们会在第15章和第16章回过来讨论这个问题)。但是他们选择了现在这种方式,因为这种方式自然且能避免语言变得更复杂。此外,大多数Java程序员都已经熟悉了具有一个抽象方法的接口的理念(例如事件处理)。试试看测验3.3,测试一下你对哪里可以使用Lambda这个知识点的掌握情况。

测验3.3:在哪里可以使用Lambda?

以下哪些是使用Lambda表达式的有效方式?

(1)
```
execute(() -> {});
public void execute(Runnable r){
    r.run();
}
```

(2)
```
public Callable<String> fetch() {
    return () -> "Tricky example  ;-)";
}
```

(3)
```
Predicate<Apple> p = (Apple a) -> a.getWeight();
```

答案:只有1和2是有效的。

第一个例子有效,是因为Lambda`() -> {}`具有签名`() -> void`,这和Runnable中的抽象方法run的签名相匹配。请注意,此代码运行后什么都不会做,因为Lambda是空的!

第二个例子也是有效的。事实上,fetch方法的返回类型是`Callable<String>`。`Callable<String>`基本上就定义了一个方法,签名是`() -> String`,其中T被String代替了。因为Lambda`() -> "Trickyexample;-)"`的签名是`() -> String`,所以在这个上下文中可以使用Lambda。

第三个例子无效,因为Lambda表达式`(Apple a) -> a.getWeight()`的签名是`(Apple) -> Integer`,这和`Predicate<Apple>:(Apple) -> boolean`中定义的test方法的签名不同。

@FunctionalInterface又是怎么回事?

如果你去看看新的Java API,会发现函数式接口带有@FunctionalInterface的标注(3.4节中会深入研究函数式接口,并会给出一个长长的列表)。这个标注用于表示该接口会设计成一个函数式接口。如果你用@FunctionalInterface定义了一个接口,而它却不是函数式接口的话,编译器将返回一个提示原因的错误。例如,错误消息可能是“Multiple non-overriding abstract methods found in interface Foo”,表明存在多个抽象方法。请注意,@FunctionalInterface不是必需的,但对于为此设计的接口而言,使用它是比较好的做法。它就像是@Override标注表示方法被重写了。

3.3 把 Lambda 付诸实践：环绕执行模式

让我们通过一个例子，看看在实践中如何利用Lambda和行为参数化来让代码更为灵活，更为简洁。资源处理（例如处理文件或数据库）时一个常见的模式就是打开一个资源，做一些处理，然后关闭资源。这个设置和清理阶段总是很类似，并且会围绕着执行处理的那些重要代码。这就是所谓的环绕执行（execute around）模式，如图3-2所示。例如，在以下代码中，高亮显示的就是从一个文件中读取一行所需的模板代码（注意你使用了Java 7中的带资源的`try`语句，它已经简化了代码，因为你不需要显式地关闭资源了）：

```
public static String processFile() throws IOException {
    try (BufferedReader br =
            new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

这就是做有用工作的那行代码

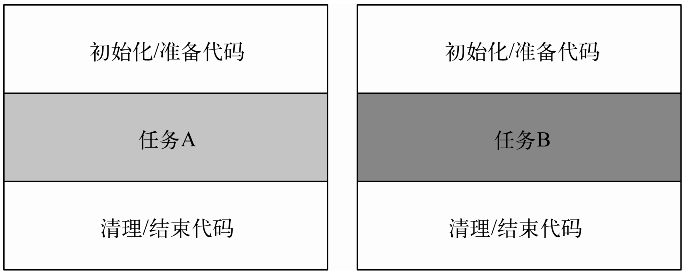


图3-2 任务A和任务B周围都环绕着进行准备/清理的同一段冗余代码

3.3.1 第 1 步：记得行为参数化

现在这段代码是有局限的。你只能读文件的第一行。如果你想要返回头两行，甚至是返回使用最频繁的词，该怎么办呢？在理想的情况下，你要重用执行设置和清理的代码，并告诉`processFile`方法对文件执行不同的操作。这听起来是不是很耳熟？是的，你需要把`processFile`的行为参数化。你需要一种方法把行为传递给`processFile`，以便它可以利用`BufferedReader`执行不同的行为。

传递行为正是Lambda的拿手好戏。那要是想一次读两行，这个新的`processFile`方法看起来又该是什么样的呢？基本上，你需要一个接收`BufferedReader`并返回`String`的Lambda。例如，下面就是从`BufferedReader`中打印两行的写法：

```
String result = processFile((BufferedReader br) ->
                            br.readLine() + br.readLine());
```

3.3.2　第 2 步：使用函数式接口来传递行为

我们前面解释过了，Lambda仅可用于上下文是函数式接口的情况。你需要创建一个能匹配BufferedReader -> String，还可以抛出IOException异常的接口。让我们把这一接口叫作BufferedReaderProcessor吧。

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

现在你就可以把这个接口作为新的processFile方法的参数了：

```
public static String processFile(BufferedReaderProcessor p) throws
        IOException {
   …
}
```

3.3.3　第 3 步：执行一个行为

任何BufferedReader -> String形式的Lambda都可以作为参数来传递，因为它们符合BufferedReaderProcessor接口中定义的process方法的签名。现在你只需要一种方法在processFile主体内执行Lambda所代表的代码。请记住，Lambda表达式允许你直接内联，为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。因此，你可以在processFile主体内，对得到的BufferedReaderProcessor对象调用process方法执行处理：

```
public static String processFile(BufferedReaderProcessor p) throws
        IOException {
    try (BufferedReader br =
            new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);        <—— 处理BufferedReader
    }                                     对象
}
```

3.3.4　第 4 步：传递 Lambda

现在你就可以通过传递不同的Lambda重用processFile方法，并以不同的方式处理文件了。

处理一行：

```
String oneLine =
    processFile((BufferedReader br) -> br.readLine());
```

处理两行：

```
String twoLines =
    processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

图3-3总结了所采取的使pocessFile方法更灵活的四个步骤。

```
public static String processFile() throws IOException {              ❶
    try (BufferedReader br =
            new BufferedReader(new FileReader("data.txt"))){
        return br.readLine();
    }
}
```

```
public interface BufferedReaderProcessor {                           ❷
    String process(BufferedReader b) throws IOException;
}

public static String processFile(BufferedReaderProcessor p) throws
IOException {
   …
}
```

```
public static String processFile(BufferedReaderProcessor p)          ❸
throws IOException {
    try (BufferedReader br=
            new BufferedReader(new FileReader("data.txt"))){
       return p.process(br);
   }
}
```

```
String oneLine = processFile((BufferedReader br) ->                   ❹
                                 br.readLine());

String twoLines = processFile((BufferedReader br) ->
                                 br.readLine() + br.readLine());

..
```

图3-3　应用环绕执行模式所采取的四个步骤

我们已经展示了如何利用函数式接口来传递Lambda，但你还是得定义你自己的接口。在下一节中，我们会探讨Java 8中加入的新接口，你可以重用它来传递多个不同的Lambda。

3.4　使用函数式接口

就像你在3.2.1节中学到的，函数式接口定义且只定义了一个抽象方法。函数式接口很有用，因为抽象方法的签名可以描述Lambda表达式的签名。函数式接口的抽象方法的签名称为函数描述符。所以为了应用不同的Lambda表达式，你需要一套能够描述常见函数描述符的函数式接口。Java API中已经有了几个函数式接口，比如你在3.2节中见到的Comparable、Runnable和Callable。

Java 8的库设计师帮你在java.util.function包中引入了几个新的函数式接口。我们接下来会介绍Predicate、Consumer和Function，更完整的列表可见本节结尾处的表3-2。

3.4.1　Predicate

java.util.function.Predicate<T>接口定义了一个名叫test的抽象方法，它接受泛型T对象，并返回一个boolean。这恰恰和你先前创建的一样，现在就可以直接使用了。在你需要表示一个涉及类型T的布尔表达式时，就可以使用这个接口。比如，你可以定义一个接受String对象的Lambda表达式，如下所示。

代码清单3-2　使用Predicate

```
@FunctionalInterface
public interface Predicate<T>{
    boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T s: list){
        if(p.test(s)){
            results.add(s);
        }
    }
    return results;
}

Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

如果你去查Predicate接口的Javadoc说明，可能会注意到诸如and和or等其他方法。现在你不用太计较这些，我们会在3.8节讨论。

3.4.2　Consumer

java.util.function.Consumer<T>定义了一个名叫accept的抽象方法，它接受泛型T的对象，没有返回（void）。你如果需要访问类型T的对象，并对其执行某些操作，就可以使用这个接口。比如，你可以用它来创建一个forEach方法，接受一个Integers的列表，并对其中每个元素执行操作。在下面的代码中，你就可以使用这个forEach方法，并配合Lambda来打印列表中的所有元素。

代码清单3-3　使用Consumer

```
@FunctionalInterface
public interface Consumer<T>{
    void accept(T t);
}
public static <T> void forEach(List<T> list, Consumer<T> c){
```

```
        for(T i: list){
            c.accept(i);
        }
    }

    forEach(
             Arrays.asList(1,2,3,4,5),
            (Integer i) -> System.out.println(i)    <—— Lambda是Consumer中accept方法的实现
           );
```


3.4.3 Function

java.util.function.Function<T, R>接口定义了一个叫作apply的方法，它接受一个泛型T的对象，并返回一个泛型R的对象。如果你需要定义一个Lambda，将输入对象的信息映射到输出，就可以使用这个接口（比如提取苹果的重量，或把字符串映射为它的长度）。在下面的代码中，我们向你展示如何利用它来创建一个map方法，以将一个String列表映射到包含每个String长度的Integer列表。

代码清单3-4 使用Function

```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T t);
}
public static <T, R> List<R> map(List<T> list,
                                 Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T s: list){
        result.add(f.apply(s));
    }
    return result;
}
// [7, 2, 6]
List<Integer> l = map(
                       Arrays.asList("lambdas","in","action"),
                       (String s) -> s.length()      <—— Lambda是Function接口的apply方法的实现
               );
```

原始类型特化

我们介绍了三个泛型函数式接口：Predicate<T>、Consumer<T>和Function<T,R>。还有些函数式接口专为某些类型而设计。

回顾一下：Java类型要么是引用类型（比如Byte、Integer、Object、List），要么是原始类型（比如int、double、byte、char）。但是泛型（比如Consumer<T>中的T）只能绑定到引用类型。这是由泛型内部的实现方式造成的。[①]因此，在Java里有一个将原始类型转换为对应的引用类型的机制。这个机制叫作装箱（boxing）。相反的操作，也就是将引用类型转换为对应

① C#等其他语言没有这一限制。Scala等语言只有引用类型。我们会在第16章再次探讨这个问题。

的原始类型，叫作拆箱（unboxing）。Java还有一个自动装箱机制来帮助程序员执行这一任务：装箱和拆箱操作是自动完成的。比如，这就是为什么下面的代码是有效的（一个int被装箱成为Integer）：

```
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; i++){
    list.add(i);
}
```

但这在性能方面是要付出代价的。装箱后的值本质上就是把原始类型包裹起来，并保存在堆里。因此，装箱后的值需要更多的内存，并需要额外的内存搜索来获取被包裹的原始值。

Java 8为我们前面所说的函数式接口带来了一个专门的版本，以便在输入和输出都是原始类型时避免自动装箱的操作。比如，在下面的代码中，使用IntPredicate就避免了对值1000进行装箱操作，但要是用Predicate<Integer>就会把参数1000装箱到一个Integer对象中：

```
public interface IntPredicate{
    boolean test(int t);
}

IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);                          <— true（无装箱）

Predicate<Integer> oddNumbers = (Integer i) -> i % 2 == 1;
oddNumbers.test(1000);                           <— false（装箱）
```

一般来说，针对专门的输入参数类型的函数式接口的名称都要加上对应的原始类型前缀，比如DoublePredicate、IntConsumer、LongBinaryOperator、IntFunction等。Function接口还有针对输出参数类型的变种：ToIntFunction<T>、IntToDoubleFunction等。

表3-2总结了Java API中提供的最常用的函数式接口及其函数描述符。请记得这只是一个起点。如果有需要，你可以自己设计一个。请记住，(T,U) -> R的表达方式展示了应当如何思考一个函数描述符。表的左侧代表了参数类型。这里它代表一个函数，具有两个参数，分别为泛型T和U，返回类型为R。

表3-2　Java 8中的常用函数式接口

函数式接口	函数描述符	原始类型特化
Predicate<T>	T->boolean	IntPredicate,LongPredicate, DoublePredicate
Consumer<T>	T->void	IntConsumer,LongConsumer, DoubleConsumer
Function<T,R>	T->R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>

（续）

函数式接口	函数描述符	原始类型特化
Supplier<T>	()->T	BooleanSupplier,IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T->T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T,T)->T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L,R>	(L,R)->boolean	
BiConsumer<T,U>	(T,U)->void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T,U,R>	(T,U)->R	ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, ToDoubleBiFunction<T,U>

你现在已经看到了很多函数式接口，可以用于描述各种Lambda表达式的签名。为了检验你的理解程度，试试测验3.4。

测验3.4：函数式接口

对于下列函数描述符（即Lambda表达式的签名），你会使用哪些函数式接口？在表3-2中可以找到大部分答案。作为进一步练习，请构造一个可以利用这些函数式接口的有效Lambda表达式：

(1) T->R

(2) (int, int)->int

(3) T->void

(4) ()->T

(5) (T, U)->R

答案如下。

(1) Function<T,R>不错。它一般用于将类型T的对象转换为类型R的对象（比如Function<Apple, Integer>用来提取苹果的重量）。

(2) IntBinaryOperator具有唯一一个抽象方法，叫作applyAsInt，它代表的函数描述符是(int, int) -> int。

(3) Consumer<T>具有唯一一个抽象方法叫作accept，代表的函数描述符是T -> void。

(4) Supplier<T>具有唯一一个抽象方法叫作get，代表的函数描述符是()-> T。或者，Callable<T>具有唯一一个抽象方法叫作call，代表的函数描述符是() -> T。

(5) BiFunction<T, U, R>具有唯一一个抽象方法叫作apply，代表的函数描述符是(T, U) -> R。

为了总结关于函数式接口和Lambda的讨论，表3-3总结了一些使用案例、Lambda的例子，以及可以使用的函数式接口。

表3-3　Lambdas及函数式接口的例子

使用案例	Lambda 的例子	对应的函数式接口
布尔表达式	(List<String> list) -> list.isEmpty()	Predicate<List<String>>
创建对象	() -> new Apple(10)	Supplier<Apple>
消费一个对象	(Apple a) -> System.out.println(a.getWeight())	Consumer<Apple>
从一个对象中选择/提取	(String s) -> s.length()	Function<String, Integer>或 ToIntFunction<String>
合并两个值	(int a, int b) -> a * b	IntBinaryOperator
比较两个对象	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())	Comparator<Apple>或 BiFunction<Apple, Apple, Integer> 或 ToIntBiFunction<Apple, Apple>

异常、Lambda，还有函数式接口又是怎么回事呢？

请注意，任何函数式接口都不允许抛出受检异常（checked exception）。如果你需要Lambda表达式来抛出异常，有两种办法：定义一个自己的函数式接口，并声明受检异常，或者把Lambda包在一个try/catch块中。

比如，在3.3节我们介绍了一个新的函数式接口BufferedReaderProcessor，它显式声明了一个IOException：

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();
```

但是你可能是在使用一个接受函数式接口的API，比如Function<T, R>，没有办法自己创建一个（你会在下一章看到，Stream API中大量使用表3-2中的函数式接口）。这种情况下，你可以显式捕捉受检异常：

```
Function<BufferedReader, String> f = (BufferedReader b) -> {
    try {
        return b.readLine();
    }
    catch(IOException e) {
        throw new RuntimeException(e);
    }
};
```

现在你知道如何创建Lambda，在哪里以及如何使用它们了。接下来我们会介绍一些更高级的细节：编译器如何对Lambda做类型检查，以及你应当了解的规则，诸如Lambda在自身内部引用局部变量，还有和void兼容的Lambda等。你无需立即就充分理解下一节的内容，可以留待日后

再看，现在可继续看3.6节讲的方法引用。

3.5　类型检查、类型推断以及限制

当我们第一次提到Lambda表达式时，说它可以为函数式接口生成一个实例。然而，Lambda表达式本身并不包含它在实现哪个函数式接口的信息。为了全面了解Lambda表达式，你应该知道Lambda的实际类型是什么。

3.5.1　类型检查

Lambda的类型是从使用Lambda的上下文推断出来的。上下文（比如，接受它传递的方法的参数，或接受它的值的局部变量）中Lambda表达式需要的类型称为目标类型。让我们通过一个例子，看看当你使用Lambda表达式时背后发生了什么。图3-4概述了下列代码的类型检查过程。

```
List<Apple> heavierThan150g =
        filter(inventory, (Apple a) -> a.getWeight() > 150);
```

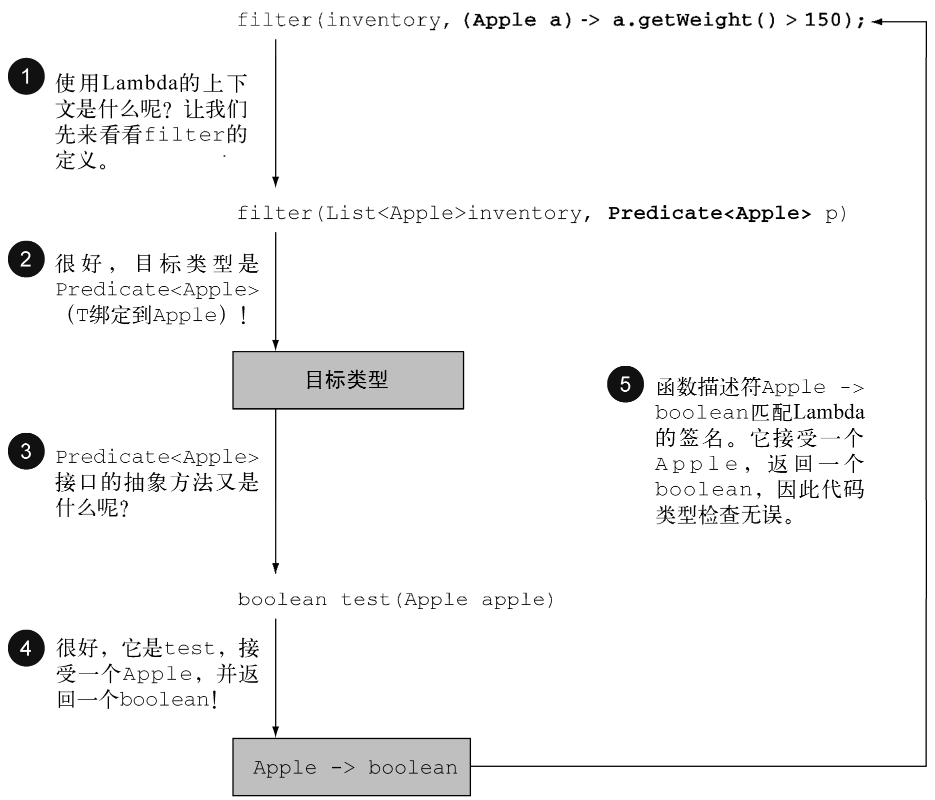


图3-4　解读Lambda表达式的类型检查过程

类型检查过程可以分解为如下所示。

- 首先，你要找出filter方法的声明。
- 第二，要求它是Predicate<Apple>（目标类型）对象的第二个正式参数。
- 第三，Predicate<Apple>是一个函数式接口，定义了一个叫作test的抽象方法。
- 第四，test方法描述了一个函数描述符，它可以接受一个Apple，并返回一个boolean。
- 最后，filter的任何实际参数都必须匹配这个要求。

这段代码是有效的，因为我们所传递的Lambda表达式也同样接受Apple为参数，并返回一个boolean。请注意，如果Lambda表达式抛出一个异常，那么抽象方法所声明的throws语句也必须与之匹配。

3.5.2 同样的 Lambda，不同的函数式接口

有了目标类型的概念，同一个Lambda表达式就可以与不同的函数式接口联系起来，只要它们的抽象方法签名能够兼容。比如，前面提到的Callable和PrivilegedAction，这两个接口都代表着什么也不接受且返回一个泛型T的函数。因此，下面两个赋值是有效的：

```
Callable<Integer> c = () -> 42;
PrivilegedAction<Integer> p = () -> 42;
```

这里，第一个赋值的目标类型是Callable<Integer>，第二个赋值的目标类型是PrivilegedAction<Integer>。

在表3-3中我们展示了一个类似的例子；同一个Lambda可用于多个不同的函数式接口：

```
Comparator<Apple> c1 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
ToIntBiFunction<Apple, Apple> c2 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
BiFunction<Apple, Apple, Integer> c3 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

菱形运算符

那些熟悉Java的演变的人会记得，Java 7中已经引入了菱形运算符（<>），利用泛型推断从上下文推断类型的思想（这一思想甚至可以追溯到更早的泛型方法）。一个类实例表达式可以出现在两个或更多不同的上下文中，并会像下面这样推断出适当的类型参数：

```
List<String> listOfStrings = new ArrayList<>();
List<Integer> listOfIntegers = new ArrayList<>();
```

特殊的void兼容规则

如果一个Lambda的主体是一个语句表达式，它就和一个返回void的函数描述符兼容（当然需要参数列表也兼容）。例如，以下两行都是合法的，尽管List的add方法返回了一个boolean，而不是Consumer上下文（T -> void）所要求的void：

```
// Predicate返回了一个boolean
Predicate<String> p = s -> list.add(s);
// Consumer返回了一个void
Consumer<String> b = s -> list.add(s);
```

到现在为止，你应该能够很好地理解在什么时候以及在哪里可以使用Lambda表达式了。它们可以从赋值的上下文、方法调用的上下文（参数和返回值），以及类型转换的上下文中获得目标类型。为了检验你的掌握情况，请试试测验3.5。

测验3.5：类型检查——为什么下面的代码不能编译呢？

你该如何解决这个问题呢？

```
Object o = () -> {System.out.println("Tricky example"); };
```

答案：Lambda表达式的上下文是`Object`（目标类型）。但`Object`不是一个函数式接口。为了解决这个问题，你可以把目标类型改成`Runnable`，它的函数描述符是`() -> void`：

```
Runnable r = () -> {System.out.println("Tricky example"); };
```

你已经见过如何利用目标类型来检查一个Lambda是否可以用于某个特定的上下文。其实，它也可以用来做一些略有不同的事：推断Lambda参数的类型。

3.5.3 类型推断

你还可以进一步简化你的代码。Java编译器会从上下文（目标类型）推断出用什么函数式接口来配合Lambda表达式，这意味着它也可以推断出适合Lambda的签名，因为函数描述符可以通过目标类型来得到。这样做的好处在于，编译器可以了解Lambda表达式的参数类型，这样就可以在Lambda语法中省去标注参数类型。换句话说，Java编译器会像下面这样推断Lambda的参数类型：[①]

```
List<Apple> greenApples =
    filter(inventory, a -> "green".equals(a.getColor()));    <-- 参数a没有显式类型
```

Lambda表达式有多个参数，代码可读性的好处就更为明显。例如，你可以这样来创建一个`Comparator`对象：

```
Comparator<Apple> c =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());    <-- 没有类型推断

Comparator<Apple> c =
    (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());    <-- 有类型推断
```

请注意，有时候显式写出类型更易读，有时候去掉它们更易读。没有什么法则说哪种更好；对于如何让代码更易读，程序员必须做出自己的选择。

① 请注意，当Lambda仅有一个类型需要推断的参数时，参数名称两边的括号也可以省略。

3.5.4　使用局部变量

我们迄今为止所介绍的所有Lambda表达式都只用到了其主体里面的参数。但Lambda表达式也允许使用自由变量（不是参数，而是在外层作用域中定义的变量），就像匿名类一样。它们被称作捕获Lambda。例如，下面的Lambda捕获了portNumber变量：

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

尽管如此，还有一点点小麻烦：关于能对这些变量做什么有一些限制。Lambda可以没有限制地捕获（也就是在其主体中引用）实例变量和静态变量。但局部变量必须显式声明为final，或事实上是final。换句话说，Lambda表达式只能捕获指派给它们的局部变量一次。（注：捕获实例变量可以被看作捕获最终局部变量this。）例如，下面的代码无法编译，因为portNumber变量被赋值两次：

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);   <-- 错误：Lambda表达式引用的局部变量必须是最终的（final）或事实上最终的
portNumber = 31337;
```

对局部变量的限制

你可能会问自己，为什么局部变量有这些限制。第一，实例变量和局部变量背后的实现有一个关键不同。实例变量都存储在堆中，而局部变量则保存在栈上。如果Lambda可以直接访问局部变量，而且Lambda是在一个线程中使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。因此，Java在访问自由局部变量时，实际上是在访问它的副本，而不是访问原始变量。如果局部变量仅仅赋值一次那就没有什么区别了——因此就有了这个限制。

第二，这一限制不鼓励你使用改变外部变量的典型命令式编程模式（我们会在以后的各章中解释，这种模式会阻碍很容易做到的并行处理）。

闭包

你可能已经听说过闭包（closure，不要和Clojure编程语言混淆）这个词，你可能会想Lambda是否满足闭包的定义。用科学的说法来说，闭包就是一个函数的实例，且它可以无限制地访问那个函数的非本地变量。例如，闭包可以作为参数传递给另一个函数。它也可以访问和修改其作用域之外的变量。现在，Java 8的Lambda和匿名类可以做类似于闭包的事情：它们可以作为参数传递给方法，并且可以访问其作用域之外的变量。但有一个限制：它们不能修改定义Lambda的方法的局部变量的内容。这些变量必须是隐式最终的。可以认为Lambda是对值封闭，而不是对变量封闭。如前所述，这种限制存在的原因在于局部变量保存在栈上，并且隐式表示它们仅限于其所在线程。如果允许捕获可改变的局部变量，就会引发造成线程不安全的新的可能性，而这是我们不想看到的（实例变量可以，因为它们保存在堆中，而堆是在线程之间共享的）。

现在，我们来介绍你会在Java 8代码中看到的另一个功能：*方法引用*。可以把它们视为某些Lambda的快捷写法。

3.6 方法引用

方法引用让你可以重复使用现有的方法定义，并像Lambda一样传递它们。在一些情况下，比起使用Lambda表达式，它们似乎更易读，感觉也更自然。下面就是我们借助更新的Java 8 API（我们会在3.7节中更详细地讨论），用方法引用写的一个排序的例子：

先前：

```
inventory.sort((Apple a1, Apple a2)
                -> a1.getWeight().compareTo(a2.getWeight()));
```

之后（使用方法引用和`java.util.Comparator.comparing`）：

```
inventory.sort(comparing(Apple::getWeight));    <— 你的第一个方法引用！
```

3.6.1 管中窥豹

你为什么应该关心方法引用？方法引用可以被看作仅仅调用特定方法的Lambda的一种快捷写法。它的基本思想是，如果一个Lambda代表的只是“直接调用这个方法”，那最好还是用名称来调用它，而不是去描述如何调用它。事实上，方法引用就是让你根据已有的方法实现来创建Lambda表达式。但是，显式地指明方法的名称，你的代码的*可读性会更好*。它是如何工作的呢？当你需要使用方法引用时，目标引用放在分隔符`::`前，方法的名称放在后面。例如，`Apple::getWeight`就是引用了`Apple`类中定义的方法`getWeight`。请记住，不需要括号，因为你没有实际调用这个方法。方法引用就是Lambda表达式`(Apple a) -> a.getWeight()`的快捷写法。表3-4给出了Java 8中方法引用的其他一些例子。

表3-4 Lambda及其等效方法引用的例子

Lambda	等效的方法引用
`(Apple a) -> a.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`

你可以把方法引用看作针对仅仅涉及单一方法的Lambda的语法糖，因为你表达同样的事情时要写的代码更少了。

如何构建方法引用

方法引用主要有三类。

(1) *指向静态方法*的方法引用（例如`Integer`的`parseInt`方法，写作`Integer::parseInt`）。

(2) 指向任意类型实例方法的方法引用（例如String的length方法，写作String::length）。

(3) 指向现有对象的实例方法的方法引用（假设你有一个局部变量expensiveTransaction用于存放Transaction类型的对象，它支持实例方法getValue，那么你就可以写expensive-Transaction::getValue）。

第二种和第三种方法引用可能乍看起来有点儿晕。类似于String::length的第二种方法引用的思想就是你在引用一个对象的方法，而这个对象本身是Lambda的一个参数。例如，Lambda表达式(String s) -> s.toUppeCase()可以写作String::toUpperCase。但第三种方法引用指的是，你在Lambda中调用一个已经存在的外部对象中的方法。例如，Lambda表达式()->expensiveTransaction.getValue()可以写作expensiveTransaction::getValue。

依照一些简单的方子，我们就可以将Lambda表达式重构为等价的方法引用，如图3-5所示。

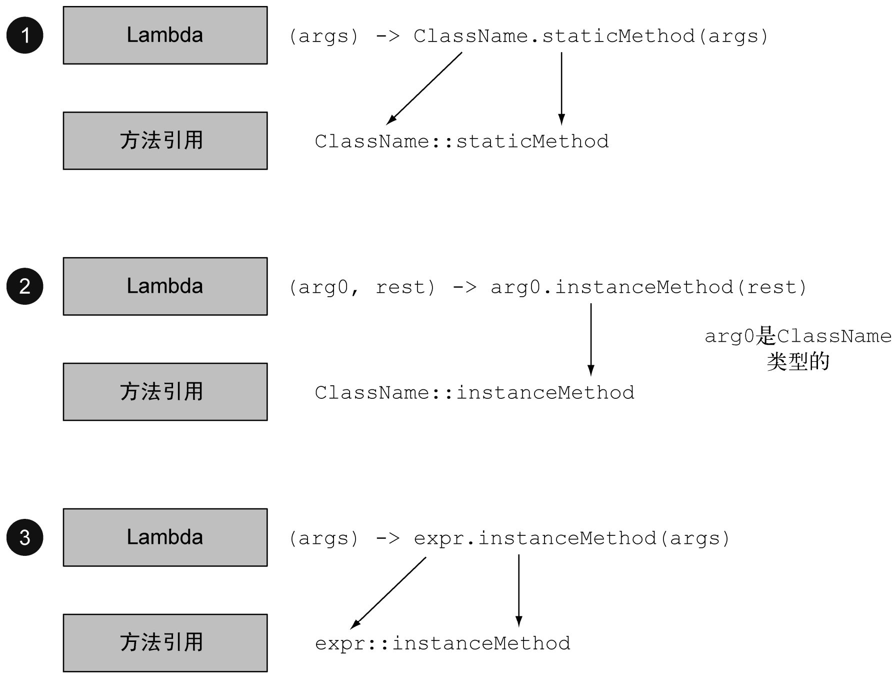


图3-5　为三种不同类型的Lambda表达式构建方法引用的办法

请注意，还有针对构造函数、数组构造函数和父类调用（super-call）的一些特殊形式的方法引用。让我们举一个方法引用的具体例子吧。比方说你想要对一个字符串的List排序，忽略大小写。List的sort方法需要一个Comparator作为参数。你在前面看到了，Comparator描述了一个具有(T, T) -> int签名的函数描述符。你可以利用String类中的compareToIgnoreCase方法来定义一个Lambda表达式（注意compareToIgnoreCase是String类中预先定义的）。

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
```

Lambda表达式的签名与`Comparator`的函数描述符兼容。利用前面所述的方法，这个例子可以用方法引用改写成下面的样子：

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort(String::compareToIgnoreCase);
```


请注意，编译器会进行一种与Lambda表达式类似的类型检查过程，来确定对于给定的函数式接口，这个方法引用是否有效：方法引用的签名必须和上下文类型匹配。

为了检验你对方法引用的理解程度，试试测验3.6吧！

测验3.6：方法引用

下列Lambda表达式的等效方法引用是什么？

(1)
```
Function<String, Integer> stringToInteger =
    (String s) -> Integer.parseInt(s);
```

(2)
```
BiPredicate<List<String>, String> contains =
    (list, element) -> list.contains(element);
```


答案如下。

(1) 这个Lambda表达式将其参数传给了`Integer`的静态方法`parseInt`。这种方法接受一个需要解析的`String`，并返回一个`Integer`。因此，可以使用图3-5中的办法❶（Lambda表达式调用静态方法）来重写Lambda表达式，如下所示：

```
Function<String, Integer> stringToInteger = Integer::parseInt;
```

(2) 这个Lambda使用其第一个参数，调用其`contains`方法。由于第一个参数是`List`类型的，你可以使用图3-5中的办法❷，如下所示：

```
BiPredicate<List<String>, String> contains = List::contains;
```

这是因为，目标类型描述的函数描述符是`(List<String>,String) -> boolean`，而`List::contains`可以被解包成这个函数描述符。

到目前为止，我们只展示了如何利用现有的方法实现和如何创建方法引用。但是你也可以对类的构造函数做类似的事情。

3.6.2　构造函数引用

对于一个现有构造函数，你可以利用它的名称和关键字`new`来创建它的一个引用：`ClassName::new`。它的功能与指向静态方法的引用类似。例如，假设有一个构造函数没有参数。它适合`Supplier`的签名`() -> Apple`。你可以这样做：

```
Supplier<Apple> c1 = Apple::new;      <—— 构造函数引用指向默认的Apple()构造函数
Apple a1 = c1.get();      <—— 调用Supplier的get方法将产生一个新的Apple
```

这就等价于：

```
Supplier<Apple> c1 = () -> new Apple();      <—— 利用默认构造函数创建Apple的Lambda表达式
Apple a1 = c1.get();      <—— 调用Supplier的get方法将产生一个新的Apple
```

如果你的构造函数的签名是`Apple(Integer weight)`，那么它就适合`Function`接口的签名，于是你可以这样写：

```
Function<Integer, Apple> c2 = Apple::new;      <—— 指向Apple(Integer weight)的构造函数引用
Apple a2 = c2.apply(110);      <—— 调用该Function函数的apply方法，并给出要求的重量，将产生一个Apple
```

这就等价于：

```
Function<Integer, Apple> c2 = (weight) -> new Apple(weight);      <—— 用要求的重量创建一个Apple的Lambda表达式
Apple a2 = c2.apply(110);      <—— 调用该Function函数的apply方法，并给出要求的重量，将产生一个新的Apple对象
```

在下面的代码中，一个由`Integer`构成的`List`中的每个元素都通过我们前面定义的类似的`map`方法传递给了`Apple`的构造函数，得到了一个具有不同重量苹果的`List`：

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);      <—— 将构造函数引用传递给map方法

public static List<Apple> map(List<Integer> list,
                              Function<Integer, Apple> f){
    List<Apple> result = new ArrayList<>();
    for(Integer e: list){
        result.add(f.apply(e));
    }
    return result;
}
```

如果你有一个具有两个参数的构造函数`Apple(String color, Integer weight)`，那么它就适合`BiFunction`接口的签名，于是你可以这样写：

```
BiFunction<String, Integer, Apple> c3 = Apple::new;      <—— 指向Apple(String color, Integer weight)的构造函数引用
Apple c3 = c3.apply("green", 110);      <—— 调用该BiFunction函数的apply方法，并给出要求的颜色和重量，将产生一个新的Apple对象
```

这就等价于：

```
BiFunction<String, Integer, Apple> c3 =
    (color, weight) -> new Apple(color, weight);   <-- 用要求的颜色和重量创建一个Apple的Lambda表达式
Apple c3 = c3.apply("green", 110);   <-- 调用该BiFunction函数的apply方法，并给出要求的颜色和重量，将产生一个新的Apple对象
```


不将构造函数实例化却能够引用它，这个功能有一些有趣的应用。例如，你可以使用`Map`来将构造函数映射到字符串值。你可以创建一个`giveMeFruit`方法，给它一个`String`和一个`Integer`，它就可以创建出不同重量的各种水果：

```
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
    map.put("apple", Apple::new);
    map.put("orange", Orange::new);
    // etc...
}
public static Fruit giveMeFruit(String fruit, Integer weight){
    return map.get(fruit.toLowerCase())   <-- 你用map得到了一个Function<Integer, Fruit>
              .apply(weight);   <-- 用Integer类型的weight参数调用Function的apply()方法将提供所要求的Fruit
}
```

为了检验你对方法和构造函数引用的理解程度，试试测验3.7吧！

测验3.7：构造函数引用

你已经看到了如何将有零个、一个、两个参数的构造函数转变为构造函数引用。那要怎么样才能对具有三个参数的构造函数，比如`Color(int, int, int)`，使用构造函数引用呢？

答案：你看，构造函数引用的语法是`ClassName::new`，那么在这个例子里面就是`Color::new`。但是你需要与构造函数引用的签名匹配的函数式接口。但是语言本身并没有提供这样的函数式接口，你可以自己创建一个：

```
public interface TriFunction<T, U, V, R>{
    R apply(T t, U u, V v);
}
```

现在你可以像下面这样使用构造函数引用了：

```
TriFunction<Integer, Integer, Integer, Color> colorFactory = Color::new;
```

我们讲了好多新内容：Lambda、函数式接口和方法引用。我们会在下一节把这一切付诸实践！

3.7 Lambda 和方法引用实战

为了给这一章还有我们讨论的所有关于Lambda的内容收个尾，我们需要继续研究开始的那个问题——用不同的排序策略给一个`Apple`列表排序，并需要展示如何把一个原始粗暴的解决方

案转变得更为简明。这会用到书中迄今讲到的所有概念和功能：行为参数化、匿名类、Lambda表达式和方法引用。我们想要实现的最终解决方案是这样的（请注意，所有源代码均可见于本书网站）：

```
inventory.sort(comparing(Apple::getWeight));
```

3.7.1　第 1 步：传递代码

你很幸运，Java 8的API已经为你提供了一个`List`可用的`sort`方法，你不用自己去实现它。那么最困难的部分已经搞定了！但是，如何把排序策略传递给`sort`方法呢？你看，`sort`方法的签名是这样的：

```
void sort(Comparator<? super E> c)
```

它需要一个`Comparator`对象来比较两个`Apple`！这就是在Java中传递策略的方式：它们必须包裹在一个对象里。我们说`sort`的*行为被参数化了*：传递给它的排序策略不同，其行为也会不同。

你的第一个解决方案看上去是这样的：

```
public class AppleComparator implements Comparator<Apple> {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
}

inventory.sort(new AppleComparator());
```

3.7.2　第 2 步：使用匿名类

你在前面看到了，你可以使用*匿名类*来改进解决方案，而不是实现一个`Comparator`却只实例化一次：

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

3.7.3　第 3 步：使用 Lambda 表达式

但你的解决方案仍然挺啰嗦的。Java 8引入了Lambda表达式，它提供了一种轻量级语法来实现相同的目标：*传递代码*。你看到了，在需要*函数式接口*的地方可以使用Lambda表达式。我们回顾一下：函数式接口就是仅仅定义一个抽象方法的接口。抽象方法的签名（称为*函数描述符*）描述了Lambda表达式的签名。在这个例子里，`Comparator`代表了函数描述符`(T, T) -> int`。因为你用的是苹果，所以它具体代表的就是`(Apple, Apple) -> int`。改进后的新解决方案看

上去就是这样的了：

```
inventory.sort((Apple a1, Apple a2)
                -> a1.getWeight().compareTo(a2.getWeight())
);
```

我们前面解释过了，Java编译器可以根据Lambda出现的上下文来推断Lambda表达式参数的类型。那么你的解决方案就可以重写成这样：

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

你的代码还能变得更易读一点吗？Comparator具有一个叫作comparing的静态辅助方法，它可以接受一个Function来提取Comparable键值，并生成一个Comparator对象（我们会在第9章解释为什么接口可以有静态方法）。它可以像下面这样用（注意你现在传递的Lambda只有一个参数：Lambda说明了如何从苹果中提取需要比较的键值）：

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

现在你可以把代码再改得紧凑一点了：

```
import static java.util.Comparator.comparing;
inventory.sort(comparing((a) -> a.getWeight()));
```

3.7.4　第 4 步：使用方法引用

前面解释过，方法引用就是替代那些转发参数的Lambda表达式的语法糖。你可以用方法引用让你的代码更简洁（假设你静态导入了java.util.Comparator.comparing）：

```
inventory.sort(comparing(Apple::getWeight));
```

恭喜你，这就是你的最终解决方案！这比Java 8之前的代码好在哪儿呢？它比较短；它的意思也很明显，并且代码读起来和问题描述差不多：“对库存进行排序，比较苹果的重量。”

3.8　复合 Lambda 表达式的有用方法

Java 8的好几个函数式接口都有为方便而设计的方法。具体而言，许多函数式接口，比如用于传递Lambda表达式的Comparator、Function和Predicate都提供了允许你进行复合的方法。这是什么意思呢？在实践中，这意味着你可以把多个简单的Lambda复合成复杂的表达式。比如，你可以让两个谓词之间做一个or操作，组合成一个更大的谓词。而且，你还可以让一个函数的结果成为另一个函数的输入。你可能会想，函数式接口中怎么可能有更多的方法呢？（毕竟，这违背了函数式接口的定义啊！）窍门在于，我们即将介绍的方法都是*默认方法*，也就是说它们不是抽象方法。我们会在第9章详谈。现在只需相信我们，等想要进一步了解默认方法以及你可以用它做什么时，再去看看第9章。

3.8.1　比较器复合

我们前面看到，你可以使用静态方法Comparator.comparing，根据提取用于比较的键值的Function来返回一个Comparator，如下所示：

```
Comparator<Apple> c = Comparator.comparing(Apple::getWeight);
```

1. 逆序

如果你想要对苹果按重量递减排序怎么办？用不着去建立另一个Comparator的实例。接口有一个默认方法reversed可以使给定的比较器逆序。因此仍然用开始的那个比较器，只要修改一下前一个例子就可以对苹果按重量递减排序：

```
inventory.sort(comparing(Apple::getWeight).reversed());  <-- 按重量递减排序
```

2. 比较器链

上面说得都很好，但如果发现有两个苹果一样重怎么办？哪个苹果应该排在前面呢？你可能需要再提供一个Comparator来进一步定义这个比较。比如，在按重量比较两个苹果之后，你可能想要按原产国排序。thenComparing方法就是做这个用的。它接受一个函数作为参数（就像comparing方法一样），如果两个对象用第一个Comparator比较之后是一样的，就提供第二个Comparator。你又可以优雅地解决这个问题了：

```
inventory.sort(comparing(Apple::getWeight)
         .reversed()                              <-- 按重量递减排序
         .thenComparing(Apple::getCountry));      <-- 两个苹果一样重时，进一步按国家排序
```

3.8.2　谓词复合

谓词接口包括三个方法：negate、and和or，让你可以重用已有的Predicate来创建更复杂的谓词。比如，你可以使用negate方法来返回一个Predicate的非，比如苹果不是红的：

```
Predicate<Apple> notRedApple = redApple.negate();  <-- 产生现有Predicate对象redApple的非
```

你可能想要把两个Lambda用and方法组合起来，比如一个苹果既是红色又比较重：

```
Predicate<Apple> redAndHeavyApple =
    redApple.and(a -> a.getWeight() > 150);    <-- 链接两个谓词来生成另一个Predicate对象
```

你可以进一步组合谓词，表达要么是重（150克以上）的红苹果，要么是绿苹果：

```
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(a -> a.getWeight() > 150)
            .or(a -> "green".equals(a.getColor()));  <-- 链接Predicate的方法来构造更复杂Predicate对象
```

这一点为什么很好呢？从简单Lambda表达式出发，你可以构建更复杂的表达式，但读起来仍然和问题的陈述差不多！请注意，and和or方法是按照在表达式链中的位置，从左向右确定优先级的。因此，a.or(b).and(c)可以看作(a || b) && c。

3.8.3 函数复合

最后，你还可以把`Function`接口所代表的Lambda表达式复合起来。`Function`接口为此配了`andThen`和`compose`两个默认方法，它们都会返回`Function`的一个实例。

`andThen`方法会返回一个函数，它先对输入应用一个给定函数，再对输出应用另一个函数。比如，假设有一个函数`f`给数字加1（`x -> x + 1`），另一个函数`g`给数字乘2，你可以将它们组合成一个函数`h`，先给数字加1，再给结果乘2：

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);
int result = h.apply(1);
```

数学上会写作`g(f(x))`或`(g o f)(x)`

这将返回4

你也可以类似地使用`compose`方法，先把给定的函数用作`compose`的参数里面给的那个函数，然后再把函数本身用于结果。比如在上一个例子里用`compose`的话，它将意味着`f(g(x))`，而`andThen`则意味着`g(f(x))`：

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);
```

数学上会写作`f(g(x))`或`(f o g)(x)`

这将返回3

图3-6说明了`andThen`和`compose`之间的区别。

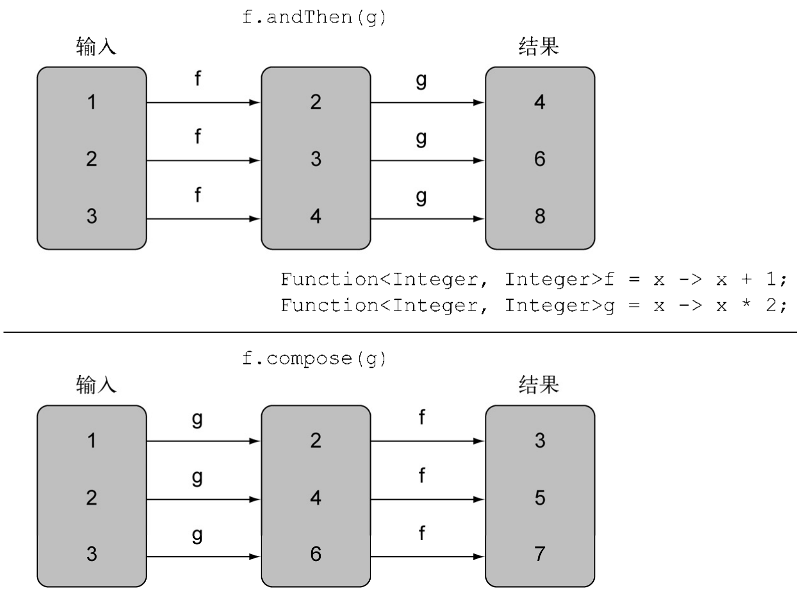


图3-6 使用`andThen`与`compose`

这一切听起来有点太抽象了。那么在实际中这有什么用呢？比方说你有一系列工具方法，对

用String表示的一封信做文本转换：

```
public class Letter{
    public static String addHeader(String text){
        return "From Raoul, Mario and Alan: " + text;
    }

    public static String addFooter(String text){
        return text + " Kind regards";
    }

    public static String checkSpelling(String text){
        return text.replaceAll("labda", "lambda");
    }
}
```

现在你可以通过复合这些工具方法来创建各种转型流水线了，比如创建一个流水线：先加上抬头，然后进行拼写检查，最后加上一个落款，如图3-7所示。

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
    = addHeader.andThen(Letter::checkSpelling)
               .andThen(Letter::addFooter);
```

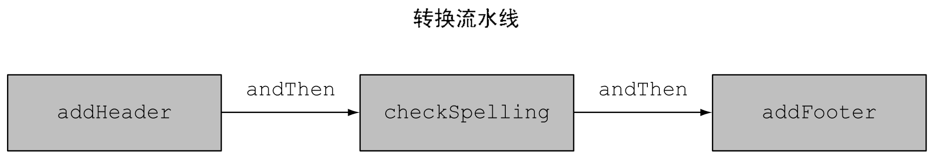


图3-7　使用andThen的转换流水线

第二个流水线可能只加抬头、落款，而不做拼写检查：

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
    = addHeader.andThen(Letter::addFooter);
```

3.9　数学中的类似思想

如果你上学的时候对数学挺拿手，那这一节就从另一个角度来谈谈Lambda表达式和函数传递的思想。你可以跳过它；书中没有任何其他内容依赖这一节，不过从另一个角度看看也挺好的。

3.9.1　积分

假设你有一个（数学，不是Java）函数f，比如说定义是

$$f(x) = x+10$$

那么，（工科学校里）经常问的一个问题就是，画在纸上之后函数下方的面积（把x轴作为基准）。

比如对于图3-8所示的区域你会写

$$\int_3^7 f(x)\mathrm{d}x \text{ 或 } \int_3^7 (x+10)\mathrm{d}x$$

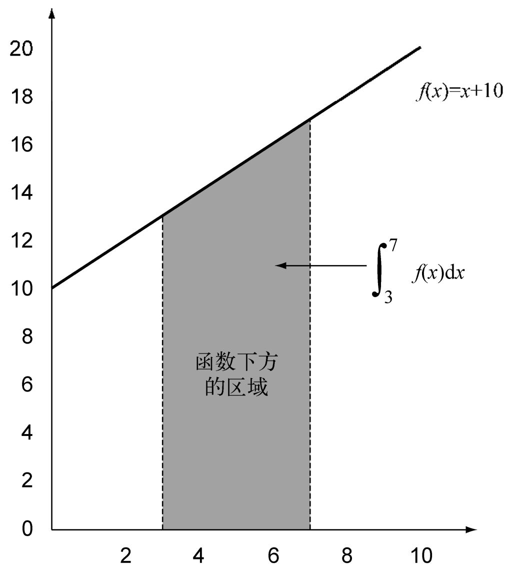


图3-8　函数`f(x)=x+10`，`x`从3到7下方的面积

在这个例子里，函数`f`是一条直线，因此你很容易通过梯形方法（画几个三角形）来算出面积：

$$1/2 \times ((3 + 10) + (7 + 10)) \times (7 - 3) = 60$$

那么这在Java里面如何表达呢？你的第一个问题是把积分号或`dy/dx`之类的换成熟悉的编程语言符号。

确实，根据第一条原则你需要一个方法，比如说叫`integrate`，它接受三个参数：一个是`f`，还有上下限（这里是3.0和7.0）。于是写在Java里就是下面这个样子，函数`f`是被传递进去的：

```
integrate(f, 3, 7)
```

请注意，你不能简单地写：

```
integrate(x + 10, 3, 7)
```

原因有二。第一，`x`的作用域不清楚；第二，这将会把`x + 10`的值而不是函数`f`传给积分。

事实上，数学上`dx`的秘密作用就是说“以`x`为自变量、结果是`x+10`的那个函数。”

3.9.2　与 Java 8 的 Lambda 联系起来

我们前面说过，Java 8的表示法`(double x) -> x + 10`（一个Lambda表达式）恰恰就是

为此设计的，因此你可以写：

```
integrate((double x) -> x + 10, 3, 7)
```

或者

```
integrate((double x) -> f(x), 3, 7)
```

或者，用前面说的方法引用，只要写：

```
integrate(C::f, 3, 7)
```

这里C是包含静态方法f的一个类。理念就是把f背后的代码传给integrate方法。

现在你可能在想如何写integrate本身了。我们还假设f是一个线性函数（直线）。你可能会写成类似数学的形式：

```
public double integrate((double -> double)f, double a, double b) {
    return (f(a)+f(b))*(b-a)/2.0
}
```

错误的Java代码！（函数的写法不能像数学里那样。）

但因为Lambda表达式只能用于接受函数式接口的地方（这里就是Function），所以你必须得写成这个样子：

```
public double integrate(DoubleFunction<Double> f, double a, double b) {
    return (f.apply(a) + f.apply(b)) * (b-a) / 2.0;
}
```

顺便提一句，有点儿可惜的是你必须写f.apply(a)，而不是像数学里面写f(a)，但Java无法摆脱"一切都是对象"的思想——它不能让函数完全独立！

3.10 小结

以下是你应从本章中学到的关键概念。

- Lambda表达式可以理解为一种匿名函数：它没有名称，但有参数列表、函数主体、返回类型，可能还有一个可以抛出的异常的列表。
- Lambda表达式让你可以简洁地传递代码。
- 函数式接口就是仅仅声明了一个抽象方法的接口。
- 只有在接受函数式接口的地方才可以使用Lambda表达式。
- Lambda表达式允许你直接内联，为函数式接口的抽象方法提供实现，并且将整个表达式作为函数式接口的一个实例。
- Java 8自带一些常用的函数式接口，放在java.util.function包里，包括Predicate<T>、Function<T,R>、Supplier<T>、Consumer<T>和BinaryOperator<T>，如表3-2所述。
- 为了避免装箱操作，对Predicate<T>和Function<T, R>等通用函数式接口的原始类型特化：IntPredicate、IntToLongFunction等。

- 环绕执行模式（即在方法所必需的代码中间，你需要执行点儿什么操作，比如资源分配和清理）可以配合Lambda提高灵活性和可重用性。
- Lambda表达式所需要代表的类型称为*目标类型*。
- 方法引用让你重复使用现有的方法实现并直接传递它们。
- `Comparator`、`Predicate`和`Function`等函数式接口都有几个可以用来结合Lambda表达式的默认方法。

Part 2

第二部分

函数式数据处理

本书第二部分深入探索了新的Stream API——它可以让你编写功能强大的代码，用声明性的方式处理数据集。学完第二部分，你将充分理解流是什么，以及如何在代码中使用它来简明而高效地处理数据集。

第4章介绍了流的概念，并解释了它与集合的异同。

第5章详细讨论了表达复杂数据处理查询可以使用的流操作。我们会谈到很多模式，如筛选、切片、查找、匹配、映射和归约。

第6章介绍了收集器——Stream API的一个功能，可以让你表达更为复杂的数据处理查询。

在第7章中，你将了解流为何可以自动并行执行，并利用多核架构的优势。此外，你还会了解到要避免的若干陷阱，以便正确而高效地使用并行流。

第4章 引入流

本章内容

- 什么是流
- 集合与流
- 内部迭代与外部迭代
- 中间操作与终端操作

集合是Java中使用最多的API。要是没有集合，还能做什么呢？几乎每个Java应用程序都会制造和处理集合。集合对于很多编程任务来说都是非常基本的：它们可以让你把数据分组并加以处理。为了解释集合是怎么工作的，想象一下你准备列出一系列菜，组成一张菜单，然后再遍历一遍，把每盘菜的热量加起来。你可能想选出那些热量比较低的菜，组成一张健康的特殊菜单。尽管集合对于几乎任何一个Java应用都是不可或缺的，但集合操作却远远算不上完美。

- 很多业务逻辑都涉及类似于数据库的操作，比如对几道菜按照类别进行分组（比如全素菜肴），或查找出最贵的菜。你自己用迭代器重新实现过这些操作多少遍？大部分数据库都允许你声明式地指定这些操作。比如，以下SQL查询语句就可以选出热量较低的菜肴名称：`SELECT name FROM dishes WHERE calorie < 400`。你看，你不需要实现如何根据菜肴的属性进行筛选（比如利用迭代器和累加器），你只需要表达你想要什么。这个基本的思路意味着，你用不着担心怎么去显式地实现这些查询语句——都替你办好了！怎么到了集合这里就不能这样了呢？
- 要是要处理大量元素又该怎么办呢？为了提高性能，你需要并行处理，并利用多核架构。但写并行代码比用迭代器还要复杂，而且调试起来也够受的！

那Java语言的设计者能做些什么，来帮助你节约宝贵的时间，让你这个程序员活得轻松一点儿呢？你可能已经猜到了，答案就是流。

4.1 流是什么

流是Java API的新成员，它允许你以声明性方式处理数据集合（通过查询语句来表达，而不是临时编写一个实现）。就现在来说，你可以把它们看成遍历数据集的高级迭代器。此外，流还

可以透明地并行处理，你无需写任何多线程代码了！我们会在第7章中详细解释流和并行化是怎么工作的。我们简单看看使用流的好处吧。下面两段代码都是用来返回低热量的菜肴名称的，并按照卡路里排序，一个是用Java 7写的，另一个是用Java 8的流写的。比较一下。不用太担心Java 8代码怎么写，我们在接下来的几节里会详细解释。

之前（Java 7）：

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){              <-- 用累加器筛选元素
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {    <-- 用匿名类对菜肴排序
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());    <-- 处理排序后的菜名列表
}
```

在这段代码中，你用了一个“垃圾变量”`lowCaloricDishes`。它唯一的作用就是作为一次性的中间容器。在Java 8中，实现的细节被放在它本该归属的库里了。

之后（Java 8）：

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
               menu.stream()
                   .filter(d -> d.getCalories() < 400)    <-- 选出400卡路里以下的菜肴
                   .sorted(comparing(Dish::getCalories))   <-- 按照卡路里排序
                   .map(Dish::getName)    <-- 提取菜肴的名称
                   .collect(toList());    <-- 将所有名称保存在List中
```

为了利用多核架构并行执行这段代码，你只需要把`stream()`换成`parallelStream()`：

```
List<String> lowCaloricDishesName =
               menu.parallelStream()
                   .filter(d -> d.getCalories() < 400)
                   .sorted(comparing(Dishes::getCalories))
                   .map(Dish::getName)
                   .collect(toList());
```

你可能会想，在调用`parallelStream`方法的时候到底发生了什么。用了多少个线程？对性能有多大提升？第7章会详细讨论这些问题。现在，你可以看出，从软件工程师的角度来看，新的方法有几个显而易见的好处。

- 代码是以声明性方式写的：说明想要完成什么（筛选热量低的菜肴）而不是说明如何实

现一个操作（利用循环和if条件等控制流语句）。你在前面的章节中也看到了，这种方法加上行为参数化让你可以轻松应对变化的需求：你很容易再创建一个代码版本，利用Lambda表达式来筛选高卡路里的菜肴，而用不着去复制粘贴代码。

- 你可以把几个基础操作链接起来，来表达复杂的数据处理流水线（在filter后面接上sorted、map和collect操作，如图4-1所示），同时保持代码清晰可读。filter的结果被传给了sorted方法，再传给map方法，最后传给collect方法。

因为filter、sorted、map和collect等操作是与具体线程模型无关的高层次构件，所以它们的内部实现可以是单线程的，也可能透明地充分利用你的多核架构！在实践中，这意味着你用不着为了让某些数据处理任务并行而去操心线程和锁了，Stream API都替你做好了！

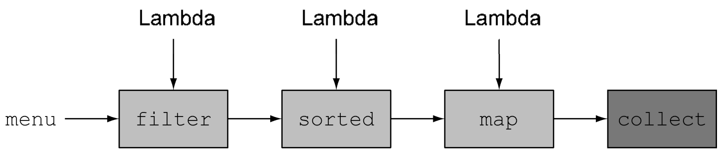


图4-1　将流操作链接起来构成流的流水线

新的Stream API表达能力非常强。比如在读完本章以及第5章、第6章之后，你就可以写出像下面这样的代码：

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

我们在第6章中解释这个例子。简单来说就是，按照Map里面的类别对菜肴进行分组。比如，Map可能包含下列结果：

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

想想要是改用循环这种典型的指令型编程方式该怎么实现吧。别浪费太多时间了。拥抱这一章和接下来几章中强大的流吧！

其他库：Guava、Apache和lambdaj

为了给Java程序员提供更好的库操作集合，前人已经做过了很多尝试。比如，Guava就是谷歌创建的一个很流行的库。它提供了multimaps和multisets等额外的容器类。Apache Commons Collections库也提供了类似的功能。最后，本书作者Mario Fusco编写的lambdaj受到函数式编程的启发，也提供了很多声明性操作集合的工具。

如今Java 8自带了官方库，可以以更加声明性的方式操作集合了。

总结一下，Java 8中的Stream API可以让你写出这样的代码：

- 声明性——更简洁，更易读

- 可复合——更灵活
- 可并行——性能更好

在本章剩下的部分和下一章中，我们会使用这样一个例子：一个menu，它只是一张菜肴列表。

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Dish类的定义是：

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    @Override
    public String toString() {
        return name;
    }
```

```
    public enum Type { MEAT, FISH, OTHER }
}
```

现在就来仔细探讨一下怎么使用Stream API。我们会用流与集合做类比，做点儿铺垫。下一章会详细讨论可以用来表达复杂数据处理查询的流操作。我们会谈到很多模式，如筛选、切片、查找、匹配、映射和归约，还会提供很多测验和练习来加深你的理解。

接下来，我们会讨论如何创建和操纵数字流，比如生成一个偶数流，或是勾股数流。最后，我们会讨论如何从不同的源（比如文件）创建流。还会讨论如何生成一个具有无穷多元素的流——这用集合肯定是搞不定了！

4.2　流简介

要讨论流，我们先来谈谈集合，这是最容易上手的方式了。Java 8中的集合支持一个新的`stream`方法，它会返回一个流（接口定义在`java.util.stream.Stream`里）。你在后面会看到，还有很多其他的方法可以得到流，比如利用数值范围或从I/O资源生成流元素。

那么，流到底是什么呢？简短的定义就是“从支持数据处理操作的源生成的元素序列”。让我们一步步剖析这个定义。

- *元素序列*——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素（如`ArrayList`与`LinkedList`）。但流的目的在于表达计算，比如你前面见到的`filter`、`sorted`和`map`。集合讲的是数据，流讲的是计算。我们会在后面几节中详细解释这个思想。
- *源*——流会使用一个提供数据的源，如集合、数组或输入/输出资源。请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元素顺序与列表一致。
- *数据处理操作*——流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如`filter`、`map`、`reduce`、`find`、`match`、`sort`等。流操作可以顺序执行，也可并行执行。

此外，流操作有两个重要的特点。

- *流水线*——很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。这让我们下一章中的一些优化成为可能，如*延迟*和*短路*。流水线的操作可以看作对数据源进行数据库式查询。
- *内部迭代*——与使用迭代器显式迭代的集合不同，流的迭代操作是在背后进行的。我们在第1章中简要地提到了这个思想，下一节会再谈到它。

让我们来看一段能够体现所有这些概念的代码：

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
```

从menu获得流（菜肴列表）

建立操作流水线：首先选出高热量的菜肴

```
                            .map(Dish::getName)          <—— 获取菜名
只选择头三个 ——>             .limit(3)
将结果保存在另一个List中 ——>  .collect(toList());
        System.out.println(threeHighCaloricDishNames);   <—— 结果是[pork, beef, chicken]
```

在本例中，我们先是对menu调用stream方法，由菜单得到一个流。数据源是菜肴列表（菜单），它给流提供一个元素序列。接下来，对流应用一系列数据处理操作：filter、map、limit和collect。除了collect之外，所有这些操作都会返回另一个流，这样它们就可以接成一条流水线，于是就可以看作对源的一个查询。最后，collect操作开始处理流水线，并返回结果（它和别的操作不一样，因为它返回的不是流，在这里是一个List）。在调用collect之前，没有任何结果产生，实际上根本就没有从menu里选择元素。你可以这么理解：链中的方法调用都在排队等待，直到调用collect。图4-2显示了流操作的顺序：filter、map、limit、collect，每个操作简介如下。

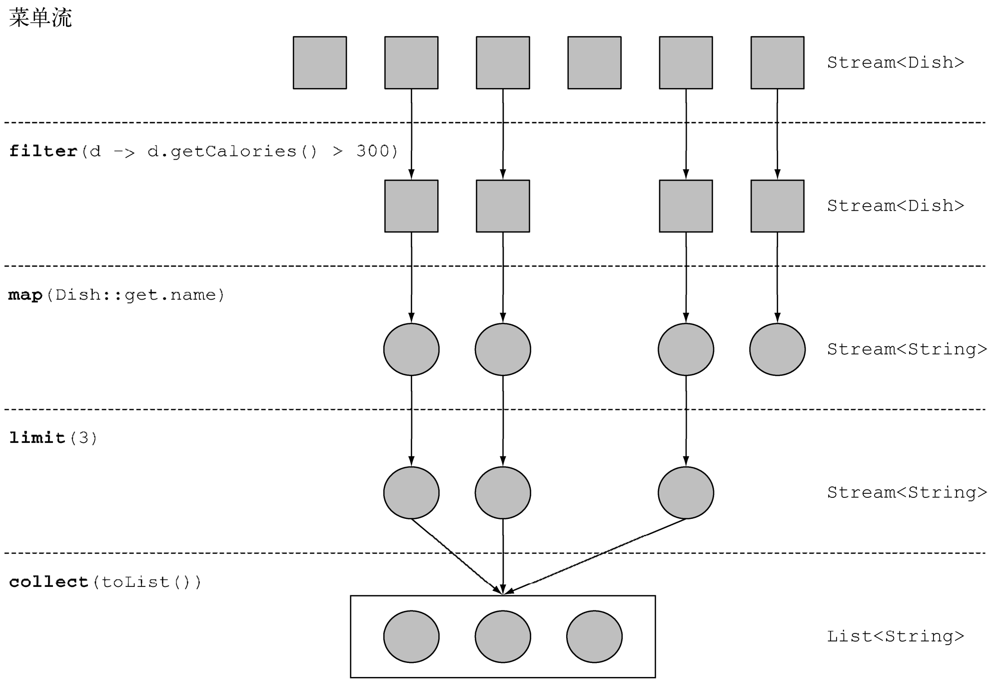


图4-2　使用流来筛选菜单，找出三个高热量菜肴的名字

- filter——接受Lambda，从流中排除某些元素。在本例中，通过传递lambda d -> d.getCalories() > 300，选择出热量超过300卡路里的菜肴。
- map——接受一个Lambda，将元素转换成其他形式或提取信息。在本例中，通过传递方法引用Dish::getName，相当于Lambda d -> d.getName()，提取了每道菜的菜名。

- `limit`——截断流，使其元素不超过给定数量。
- `collect`——将流转换为其他形式。在本例中，流被转换为一个列表。它看起来有点儿像变魔术，我们在第6章中会详细解释`collect`的工作原理。现在，你可以把`collect`看作能够接受各种方案作为参数，并将流中的元素累积成为一个汇总结果的操作。这里的`toList()`就是将流转换为列表的方案。

注意看，我们刚刚解释的这段代码，与逐项处理菜单列表的代码有很大不同。首先，我们使用了声明性的方式来处理菜单数据，即你说的对这些数据*需要做什么*："查找热量最高的三道菜的菜名。"你并没有去实现筛选（`filter`）、提取（`map`）或截断（`limit`）功能，Streams库已经自带了。因此，Stream API在决定如何优化这条流水线时更为灵活。例如，筛选、提取和截断操作可以一次进行，并在找到这三道菜后立即停止。我们会在下一章介绍一个能体现这一点的例子。

在进一步介绍能对流做什么操作之前，先让我们回过头来看看Collection API和新的Stream API的思想有何不同。

4.3　流与集合

Java现有的集合概念和新的流概念都提供了接口，来配合代表元素型有序值的数据接口。所谓*有序*，就是说我们一般是按顺序取用值，而不是随机取用的。那这两者有什么区别呢？

我们先来打个直观的比方吧。比如说存在DVD里的电影，这就是一个集合（也许是字节，也许是帧，这个无所谓），因为它包含了整个数据结构。现在再来想想在互联网上通过*视频流*看同样的电影。现在这是一个流（字节流或帧流）。流媒体视频播放器只要提前下载用户观看位置的那几帧就可以了，这样不用等到流中大部分值计算出来，你就可以显示流的开始部分了（想想观看直播足球赛）。特别要注意，视频播放器可能没有将整个流作为集合，保存所需要的内存缓冲区——而且要是非得等到最后一帧出现才能开始看，那等待的时间就太长了。出于实现的考虑，你也可以让视频播放器把流的一部分*缓存*在集合里，但和概念上的差异不是一回事。

粗略地说，集合与流之间的差异就在于*什么时候*进行计算。集合是一个内存中的数据结构，它包含数据结构中目前*所有*的值——集合中的每个元素都得先算出来才能添加到集合中。（你可以往集合里加东西或者删东西，但是不管什么时候，集合中的每个元素都是放在内存里的，元素都得先算出来才能成为集合的一部分。）

相比之下，流则是在概念上固定的数据结构（你不能添加或删除元素），其元素则是*按需计算*的。 这对编程有很大的好处。在第6章中，我们将展示构建一个质数流（2, 3, 5, 7, 11, …）有多简单，尽管质数有无穷多个。这个思想就是用户仅仅从流中提取需要的值，而这些值——在用户看不见的地方——只会*按需*生成。这是一种生产者－消费者的关系。从另一个角度来说，流就像是一个延迟创建的集合：只有在消费者要求的时候才会计算值（用管理学的话说这就是需求驱动，甚至是实时制造)。

与此相反，集合则是急切创建的（供应商驱动：先把仓库装满，再开始卖，就像那些昙花一

现的圣诞新玩意儿一样）。以质数为例，要是想创建一个包含所有质数的集合，那这个程序算起来就没完没了了，因为总有新的质数要算，然后把它加到集合里面。当然这个集合是永远也创建不完的，消费者这辈子都见不着了。

图4-3用DVD对比在线流媒体的例子展示了流和集合之间的差异。

另一个例子是用浏览器进行互联网搜索。假设你搜索的短语在Google或是网店里面有很多匹配项。你用不着等到所有结果和照片的集合下载完，而是得到一个流，里面有最好的10个或20个匹配项，还有一个按钮来查看下面10个或20个。当你作为消费者点击“下面10个”的时候，供应商就按需计算这些结果，然后再送回你的浏览器上显示。

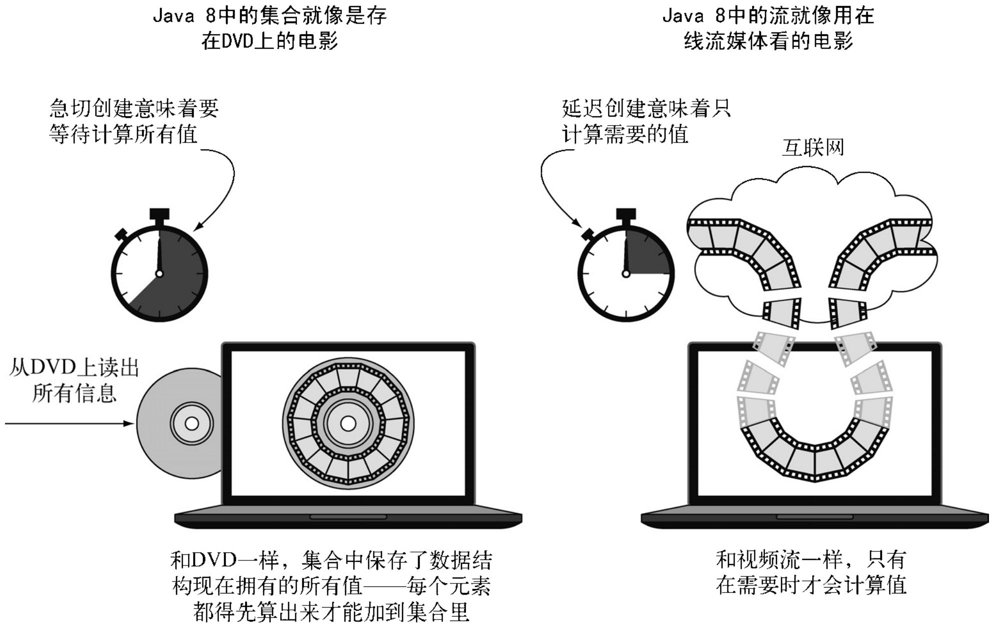


图4-3　流与集合

4.3.1　只能遍历一次

请注意，和迭代器类似，流只能遍历一次。遍历完之后，我们就说这个流已经被消费掉了。你可以从原始数据源那里再获得一个新的流来重新遍历一遍，就像迭代器一样（这里假设它是集合之类的可重复的源，如果是I/O通道就没戏了）。例如，以下代码会抛出一个异常，说流已被消费掉了：

```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```

打印标题中的每个单词

java.lang.IllegalStateException:流已被操作或关闭

所以要记得，流只能消费一次！

哲学中的流和集合

对于喜欢哲学的读者，你可以把流看作在时间中分布的一组值。相反，集合则是空间（这里就是计算机内存）中分布的一组值，在一个时间点上全体存在——你可以使用迭代器来访问for-each循环中的内部成员。

集合和流的另一个关键区别在于它们遍历数据的方式。

4.3.2　外部迭代与内部迭代

使用Collection接口需要用户去做迭代（比如用for-each），这称为外部迭代。 相反，Streams库使用内部迭代——它帮你把迭代做了，还把得到的流值存在了某个地方，你只要给出一个函数说要干什么就可以了。下面的代码列表说明了这种区别。

代码清单4-1　集合：用for-each循环外部迭代

```
List<String> names = new ArrayList<>();
for(Dish d: menu){                          <-- 显式顺序迭代菜单列表
    names.add(d.getName());                 <-- 提取名称并将其添加到累加器
}
```

请注意，for-each还隐藏了迭代中的一些复杂性。for-each结构是一个语法糖，它背后的东西用Iterator对象表达出来更要丑陋得多。

代码清单4-2　集合：用背后的迭代器做外部迭代

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {                 <-- 显式迭代
    Dish d = iterator.next();
    names.add(d.getName());
}
```

代码清单4-3　流：内部迭代

```
List<String> names = menu.stream()
                         .map(Dish::getName)        <-- 用getName方法参数化map，提取菜名
                         .collect(toList());        <-- 开始执行操作流水线；没有迭代！
```

让我们用一个比喻来解释内部迭代的差异和好处吧。比方说你在和你两岁的女儿索菲亚说话，希望她能把玩具收起来。

你：“索菲亚，我们把玩具收起来吧。地上还有玩具吗？”

索菲亚：“有，球。”

你：“好，把球放进盒子里。还有吗？”

索菲亚：“有，那是我的娃娃。”

你：“好，把娃娃放进盒子里。还有吗？”

索菲亚：“有，有我的书。”

你：“好，把书放进盒子里。还有吗？”

索菲亚：“没了，没有了。”

你：“好，我们收好啦。”

这正是你每天都要对Java集合做的。你外部迭代一个集合，显式地取出每个项目再加以处理。如果你只需跟索菲亚说“把地上所有的玩具都放进盒子里”就好了。内部迭代比较好的原因有二：第一，索非亚可以选择一只手拿娃娃，另一只手拿球；第二，她可以决定先拿离盒子最近的那个东西，然后再拿别的。同样的道理，内部迭代时，项目可以透明地并行处理，或者用更优化的顺序进行处理。要是用Java过去的那种外部迭代方法，这些优化都是很困难的。这似乎有点儿鸡蛋里挑骨头，但这差不多就是Java 8引入流的理由了——Streams库的内部迭代可以自动选择一种适合你硬件的数据表示和并行实现。与此相反，一旦通过写`for-each`而选择了外部迭代，那你基本上就要自己管理所有的并行问题了（自己管理实际上意味着“某个良辰吉日我们会把它并行化”或“开始了关于任务和`synchronized`的漫长而艰苦的斗争”）。Java 8需要一个类似于`Collection`却没有迭代器的接口，于是就有了`Stream`！图4-4说明了流（内部迭代）与集合（外部迭代）之间的差异。

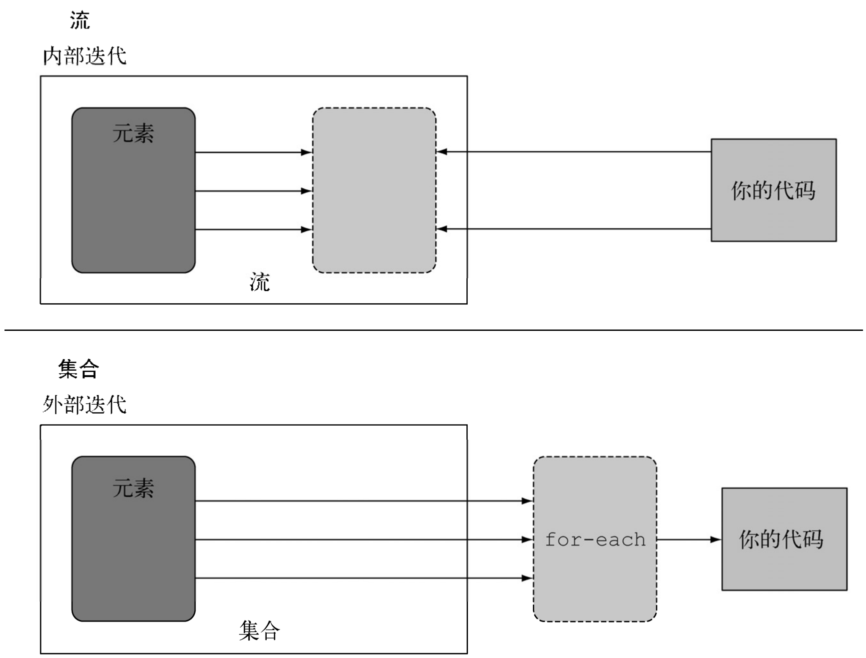


图4-4　内部迭代与外部迭代

我们已经说过了集合与流在概念上的差异，特别是流利用了内部迭代：替你把迭代做了。但是，只有你已经预先定义好了能够隐藏迭代的操作列表，例如`filter`或`map`，这个才有用。大多数这类操作都接受Lambda表达式作为参数，因此你可以用前面几章中介绍的方法来参数化其行为。Java语言的设计者给Stream API配上了一大套可以用来表达复杂数据处理查询的操作。我们现在先简要地看一下这些操作，下一章中会配上例子详细讨论。

4.4　流操作

`java.util.stream.Stream`中的`Stream`接口定义了许多操作。它们可以分为两大类。我们再来看一下前面的例子：

```
List<String> names = menu.stream()
                         .filter(d -> d.getCalories() > 300)
                         .map(Dish::getName)
                         .limit(3)
                         .collect(toList());
```

从菜单获得流

中间操作

中间操作

中间操作

将Stream转换为List

你可以看到两类操作：

- ❑ `filter`、`map`和`limit`可以连成一条流水线；
- ❑ `collect`触发流水线执行并关闭它。

可以连接起来的流操作称为中间操作，关闭流的操作称为终端操作。 图4-5中展示了这两类操作。这种区分有什么意义呢？

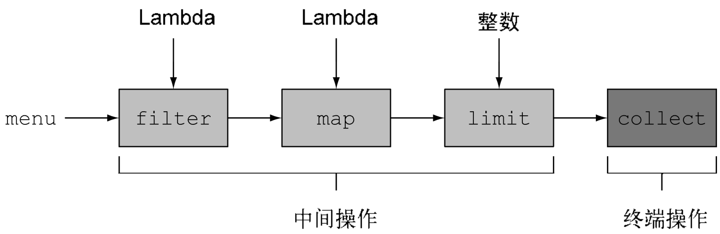


图4-5　中间操作与终端操作

4.4.1　中间操作

诸如`filter`或`sorted`等中间操作会返回另一个流。这让多个操作可以连接起来形成一个查询。重要的是，除非流水线上触发一个终端操作，否则中间操作不会执行任何处理——它们很懒。这是因为中间操作一般都可以合并起来，在终端操作时一次性全部处理。

为了搞清楚流水线中到底发生了什么，我们把代码改一改，让每个Lambda都打印出当前处理的菜肴（就像很多演示和调试技巧一样，这种编程风格要是搁在生产代码里那就吓死人了，但是学习的时候却可以直接看清楚求值的顺序）：

```
List<String> names =
    menu.stream()
        .filter(d -> {
                    System.out.println("filtering" + d.getName());
                    return d.getCalories() > 300;
                })
        .map(d -> {
                System.out.println("mapping" + d.getName());
                return d.getName();
            })
        .limit(3)
        .collect(toList());
System.out.println(names);
```

打印当前筛选的菜肴

提取菜名时打印出来

此代码执行时将打印：

```
filtering pork
mapping pork
filtering beef
mapping beef
filtering chicken
mapping chicken
[pork, beef, chicken]
```

你会发现，有好几种优化利用了流的延迟性质。第一，尽管很多菜的热量都高于300卡路里，但只选出了前三个！这是因为`limit`操作和一种称为*短路*的技巧，我们会在下一章中解释。第二，尽管`filter`和`map`是两个独立的操作，但它们合并到同一次遍历中了（我们把这种技术叫作*循环合并*）。

4.4.2 终端操作

终端操作会从流的流水线生成结果。其结果是任何不是流的值，比如`List`、`Integer`，甚至`void`。例如，在下面的流水线中，`forEach`是一个返回`void`的终端操作，它会对源中的每道菜应用一个Lambda。把`System.out.println`传递给`forEach`，并要求它打印出由`menu`生成的流中的每一个`Dish`：

```
menu.stream().forEach(System.out::println);
```

为了检验你对中间操作和终端操作的理解程度，试试测验4.1吧。

测验4.1：中间操作与终端操作

在下列流水线中，你能找出中间操作和终端操作吗？

```
long count = menu.stream()
                 .filter(d -> d.getCalories() > 300)
                 .distinct()
                 .limit(3)
                 .count();
```

答案：流水线中最后一个操作`count`返回一个`long`，这是一个非`Stream`的值。因此它是

一个终端操作。所有前面的操作，filter、distinct、limit，都是连接起来的，并返回一个Stream，因此它们是中间操作。

4.4.3　使用流

总而言之，流的使用一般包括三件事：

- 一个数据源（如集合）来执行一个查询；
- 一个中间操作链，形成一条流的流水线；
- 一个终端操作，执行流水线，并能生成结果。

流的流水线背后的理念类似于构建器模式。[①]在构建器模式中有一个调用链用来设置一套配置（对流来说这就是一个中间操作链），接着是调用built方法（对流来说就是终端操作）。

为方便起见，表4-1和表4-2总结了你前面在代码例子中看到的中间流操作和终端流操作。请注意这并不能涵盖Stream API提供的操作，你在下一章中还会看到更多。

表4-1　中间操作

操　　作	类　　型	返回类型	操作参数	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
map	中间	Stream<R>	Function<T, R>	T -> R
limit	中间	Stream<T>		
sorted	中间	Stream<T>	Comparator<T>	(T, T) -> int
distinct	中间	Stream<T>		

表4-2　终端操作

操　　作	类　　型	目　　的
forEach	终端	消费流中的每个元素并对其应用 Lambda。这一操作返回 void
count	终端	返回流中元素的个数。这一操作返回 long
collect	终端	把流归约成一个集合，比如 List、Map 甚至是 Integer。详见第 6 章

在下一章中，我们会用案例详细介绍一些可以用的流操作，让你了解可以用它们表达什么样的查询。我们会看到很多模式，比如过滤、切片、查找、匹配、映射和归约，它们可以用来表达复杂的数据处理查询。

因为第6章会非常详细地讨论收集器，所以本章和下一章仅介绍把collect()终端操作用于collect(toList())的特殊情况。这一操作会创建一个与流具有相同元素的列表。

① 见http://en.wikipedia.org/wiki/Builder_pattern。

4.5 小结

以下是你应从本章中学到的一些关键概念。

- 流是“从支持数据处理操作的源生成的一系列元素”。
- 流利用内部迭代：迭代通过`filter`、`map`、`sorted`等操作被抽象掉了。
- 流操作有两类：中间操作和终端操作。
- `filter`和`map`等中间操作会返回一个流，并可以链接在一起。可以用它们来设置一条流水线，但并不会生成任何结果。
- `forEach`和`count`等终端操作会返回一个非流的值，并处理流水线以返回结果。
- 流中的元素是按需计算的。

第 5 章

使用流

本章内容

- 筛选、切片和匹配
- 查找、匹配和归约
- 使用数值范围等数值流
- 从多个源创建流
- 无限流

在上一章中你已看到了，流让你从外部迭代转向内部迭代。 这样，你就用不着写下面这样的代码来显式地管理数据集合的迭代（外部迭代）了：

```
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.isVegetarian()){
        vegetarianDishes.add(d);
    }
}
```

你可以使用支持`filter`和`collect`操作的Stream API（内部迭代）管理对集合数据的迭代。你只需要将筛选行为作为参数传递给`filter`方法就行了。

```
import static java.util.stream.Collectors.toList;
List<Dish> vegetarianDishes =
    menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```

这种处理数据的方式很有用，因为你让Stream API管理如何处理数据。这样Stream API就可以在背后进行多种优化。此外，使用内部迭代的话，Stream API可以决定并行运行你的代码。这要是用外部迭代的话就办不到了，因为你只能用单一线程挨个迭代。

在本章中，你将会看到Stream API支持的许多操作。这些操作能让你快速完成复杂的数据查询，如筛选、切片、映射、查找、匹配和归约。接下来，我们会看看一些特殊的流：数值流、来自文件和数组等多种来源的流，最后是无限流。

5.1 筛选和切片

在本节中，我们来看看如何选择流中的元素：用谓词筛选，筛选出各不相同的元素，忽略流中的头几个元素，或将流截短至指定长度。

5.1.1 用谓词筛选

Streams接口支持filter方法（你现在应该很熟悉了）。该操作会接受一个谓词（一个返回boolean的函数）作为参数，并返回一个包括所有符合谓词的元素的流。例如，你可以像图5-1所示的这样，筛选出所有素菜，创建一张素食菜单：

```
List<Dish> vegetarianMenu = menu.stream()
                                .filter(Dish::isVegetarian)
                                .collect(toList());
```

方法引用检查菜肴是否适合素食者

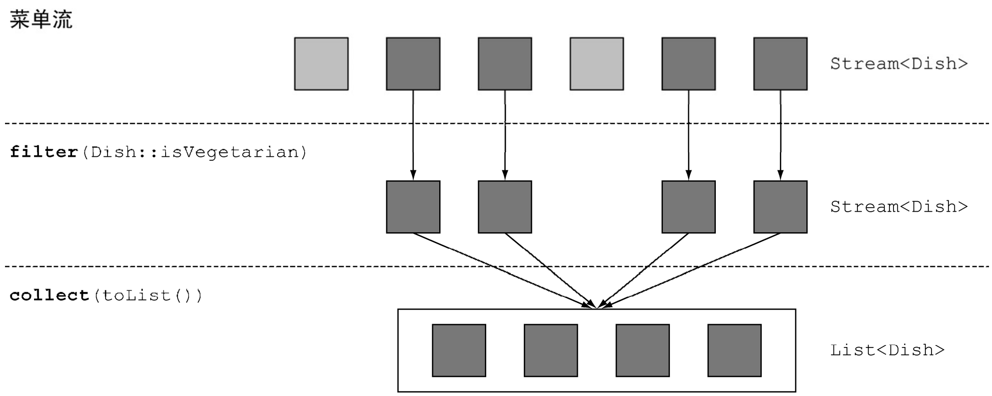


图5-1 用谓词筛选一个流

5.1.2 筛选各异的元素

流还支持一个叫作distinct的方法，它会返回一个元素各异（根据流所生成元素的hashCode和equals方法实现）的流。例如，以下代码会筛选出列表中所有的偶数，并确保没有重复。图5-2直观地显示了这个过程。

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach(System.out::println);
```

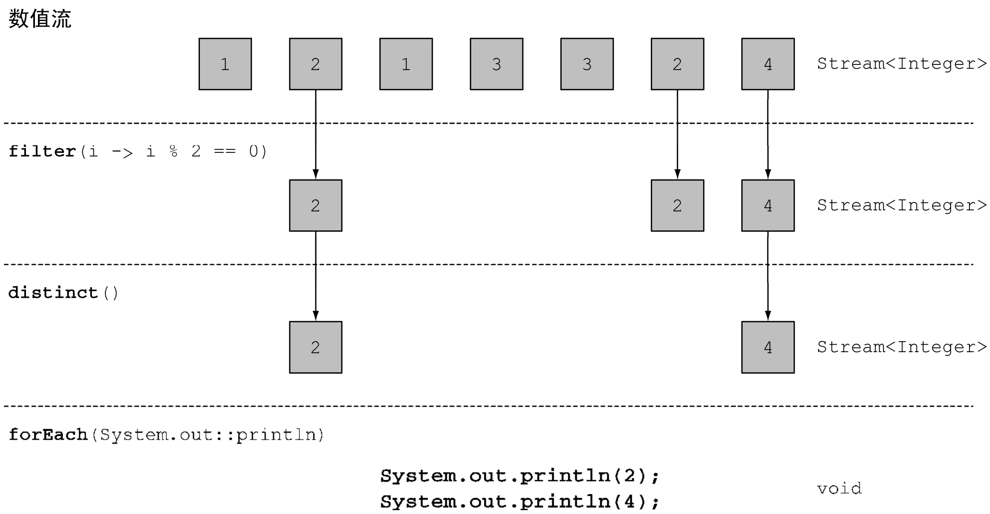


图5-2　筛选流中各异的元素

5.1.3　截短流

流支持`limit(n)`方法，该方法会返回一个不超过给定长度的流。所需的长度作为参数传递给`limit`。如果流是有序的，则最多会返回前n个元素。比如，你可以建立一个`List`，选出热量超过300卡路里的头三道菜：

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .limit(3)
                        .collect(toList());
```

图5-3展示了`filter`和`limit`的组合。你可以看到，该方法只选出了符合谓词的头三个元素，然后就立即返回了结果。

请注意`limit`也可以用在无序流上，比如源是一个`Set`。这种情况下，`limit`的结果不会以任何顺序排列。

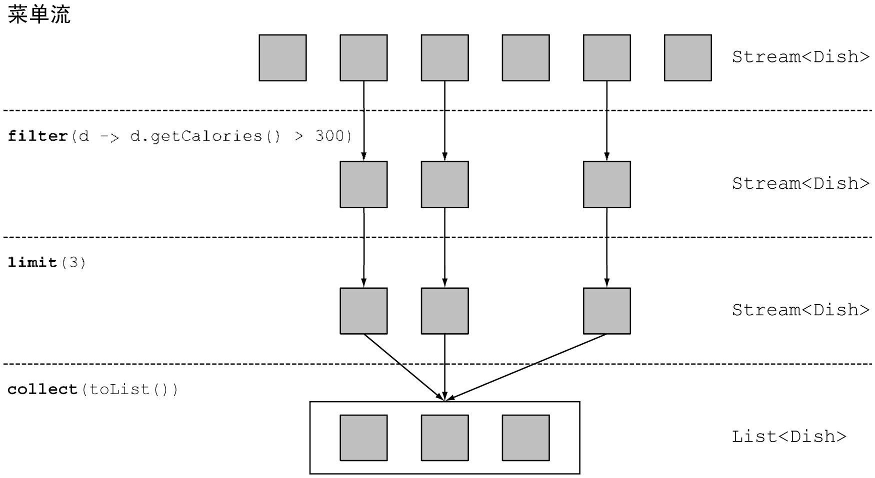


图5-3　截短流

5.1.4　跳过元素

流还支持skip(n)方法，返回一个扔掉了前n个元素的流。如果流中元素不足n个，则返回一个空流。请注意，limit(n)和skip(n)是互补的！例如，下面的代码将跳过超过300卡路里的头两道菜，并返回剩下的。图5-4展示了这个查询。

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```

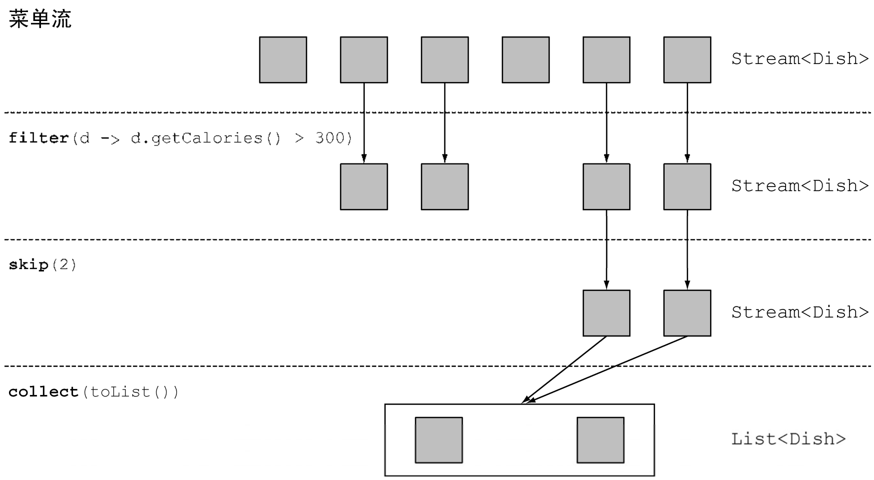


图5-4　在流中跳过元素

在我们讨论映射操作之前，在测验5.1上试试本节学过的内容吧。

测验5.1：筛选

你将如何利用流来筛选前两个荤菜呢？

答案：你可以把filter和limit复合在一起来解决这个问题，并用collect(toList())将流转换成一个列表。

```
List<Dish> dishes =
    menu.stream()
        .filter(d -> d.getType() == Dish.Type.MEAT)
        .limit(2)
        .collect(toList());
```

5.2　映射

一个非常常见的数据处理套路就是从某些对象中选择信息。比如在SQL里，你可以从表中选择一列。Stream API也通过map和flatMap方法提供了类似的工具。

5.2.1　对流中每一个元素应用函数

流支持map方法，它会接受一个函数作为参数。这个函数会被应用到每个元素上，并将其映射成一个新的元素（使用映射一词，是因为它和转换类似，但其中的细微差别在于它是“创建一个新版本”而不是去“修改”）。例如，下面的代码把方法引用Dish::getName传给了map方法，来提取流中菜肴的名称：

```
List<String> dishNames = menu.stream()
                             .map(Dish::getName)
                             .collect(toList());
```

因为getName方法返回一个String，所以map方法输出的流的类型就是Stream<String>。

让我们看一个稍微不同的例子来巩固一下对map的理解。给定一个单词列表，你想要返回另一个列表，显示每个单词中有几个字母。怎么做呢？你需要对列表中的每个元素应用一个函数。这听起来正好该用map方法去做！应用的函数应该接受一个单词，并返回其长度。你可以像下面这样，给map传递一个方法引用String::length来解决这个问题：

```
List<String> words = Arrays.asList("Java 8", "Lambdas", "In", "Action");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
```

现在让我们回到提取菜名的例子。如果你要找出每道菜的名称有多长，怎么做？你可以像下面这样，再链接上一个map：

```
List<Integer> dishNameLengths = menu.stream()
                                    .map(Dish::getName)
                                    .map(String::length)
                                    .collect(toList());
```

5.2.2 流的扁平化

你已经看到如何使用map方法返回列表中每个单词的长度了。让我们拓展一下：对于一张单词表，如何返回一张列表，列出里面各不相同的字符呢？例如，给定单词列表["Hello","World"]，你想要返回列表["H","e","l", "o","W","r","d"]。

你可能会认为这很容易，你可以把每个单词映射成一张字符表，然后调用distinct来过滤重复的字符。第一个版本可能是这样的：

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

这个方法的问题在于，传递给map方法的Lambda为每个单词返回了一个String[]（String列表）。因此，map返回的流实际上是Stream<String[]>类型的。你真正想要的是用Stream<String>来表示一个字符流。图5-5说明了这个问题。

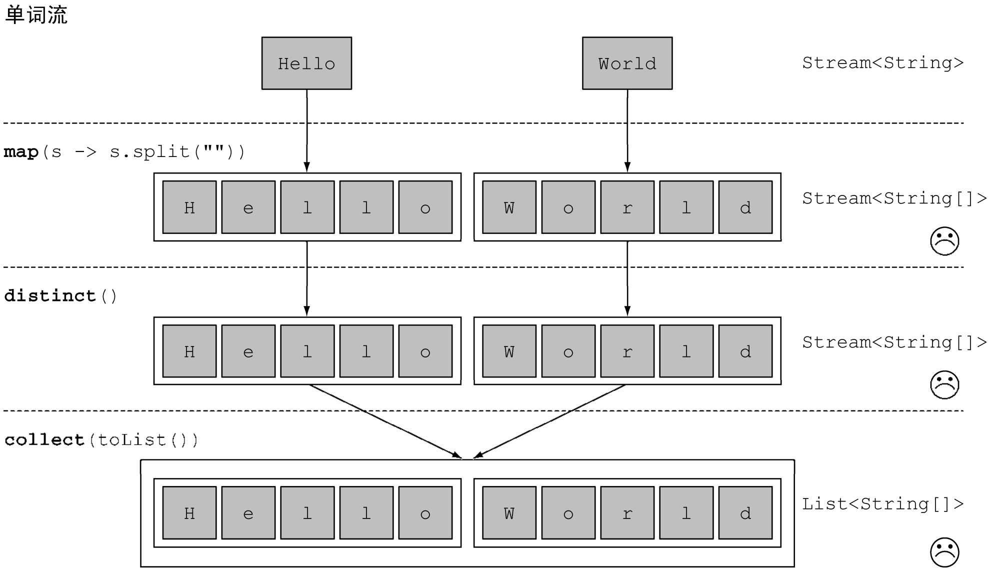


图5-5 不正确地使用map找出单词列表中各不相同的字符

幸好可以用flatMap来解决这个问题！让我们一步步看看怎么解决它。

1. 尝试使用**map**和**Arrays.stream()**

首先，你需要一个字符流，而不是数组流。有一个叫作Arrays.stream()的方法可以接受一个数组并产生一个流，例如：

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

把它用在前面的那个流水线里，看看会发生什么：

```
words.stream()
     .map(word -> word.split(""))     <-- 将每个单词转换为由其字母构成的数组
     .map(Arrays::stream)     <-- 让每个数组变成一个单独的流
     .distinct()
     .collect(toList());
```

当前的解决方案仍然搞不定！这是因为，你现在得到的是一个流的列表（更准确地说是Stream<String>）！的确，你先是把每个单词转换成一个字母数组，然后把每个数组变成了一个独立的流。

2. 使用**flatMap**

你可以像下面这样使用flatMap来解决这个问题：

```
List<String> uniqueCharacters =
    words.stream()
         .map(w -> w.split(""))     <-- 将每个单词转换为由其字母构成的数组
         .flatMap(Arrays::stream)     <-- 将各个生成流扁平化为单个流
         .distinct()
         .collect(Collectors.toList());
```

使用flatMap方法的效果是，各个数组并不是分别映射成一个流，而是映射成流的内容。所有使用map(Arrays::stream)时生成的单个流都被合并起来，即扁平化为一个流。图5-6说明了使用flatMap方法的效果。把它和图5-5中map的效果比较一下。

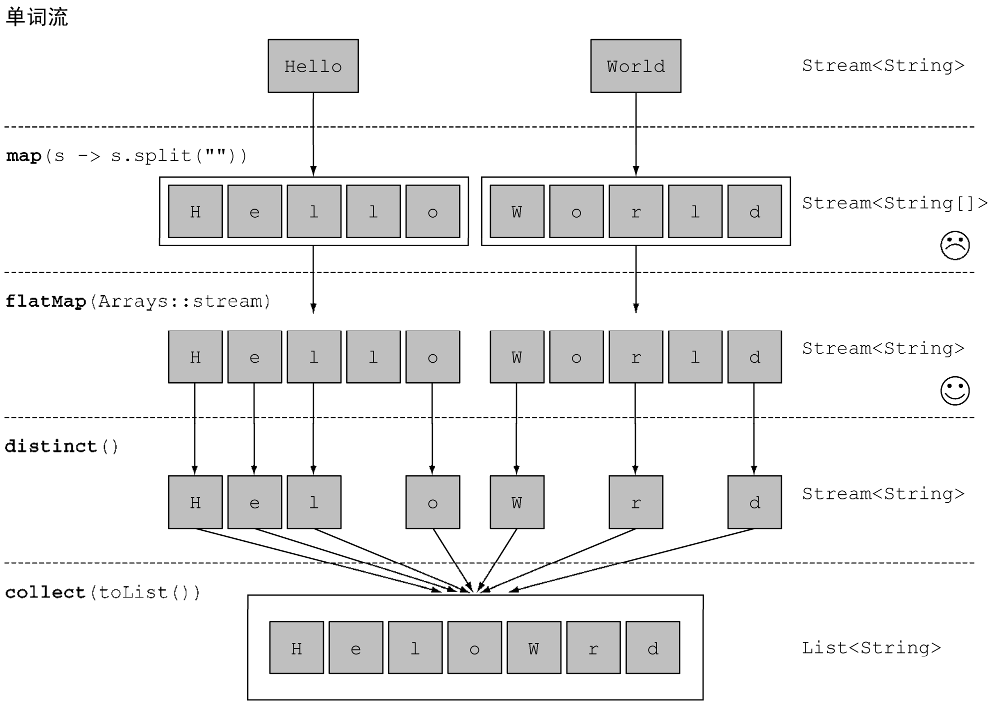


图5-6 使用flatMap找出单词列表中各不相同的字符

一言以蔽之，flatmap方法让你把一个流中的每个值都换成另一个流，然后把所有的流连接起来成为一个流。

在第10章，我们会讨论更高级的Java 8模式，比如使用新的Optional类进行null检查时会再来看看flatMap。为巩固你对于map和flatMap的理解，试试测验5.2吧。

测验5.2：映射

(1) 给定一个数字列表，如何返回一个由每个数的平方构成的列表呢？例如，给定[1, 2, 3, 4, 5]，应该返回[1, 4, 9, 16, 25]。

答案：你可以利用map方法的Lambda，接受一个数字，并返回该数字平方的Lambda来解决这个问题。

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares =
    numbers.stream()
           .map(n -> n * n)
           .collect(toList());
```

(2) 给定两个数字列表，如何返回所有的数对呢？例如，给定列表[1, 2, 3]和列表[3, 4]，应该返回[(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]。为简单起见，你可以用有两个元素的数组来代表数对。

答案：你可以使用两个map来迭代这两个列表，并生成数对。但这样会返回一个Stream-<Stream<Integer[]>>。你需要让生成的流扁平化，以得到一个Stream<Integer[]>。这正是flatMap所做的：

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
            .flatMap(i -> numbers2.stream()
                              .map(j -> new int[]{i, j})
                    )
            .collect(toList());
```

(3) 如何扩展前一个例子，只返回总和能被3整除的数对呢？例如(2, 4)和(3, 3)是可以的。

答案：你在前面看到了，filter可以配合谓词使用来筛选流中的元素。因为在flatMap操作后，你有了一个代表数对的int[]流，所以你只需要一个谓词来检查总和是否能被3整除就可以了：

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
            .flatMap(i ->
                        numbers2.stream()
                                .filter(j -> (i + j) % 3 == 0)
                                .map(j -> new int[]{i, j})
                    )
            .collect(toList());
```

其结果是[(2, 4), (3, 3)]。

5.3　查找和匹配

另一个常见的数据处理套路是看看数据集中的某些元素是否匹配一个给定的属性。Stream API通过allMatch、anyMatch、noneMatch、findFirst和findAny方法提供了这样的工具。

5.3.1　检查谓词是否至少匹配一个元素

anyMatch方法可以回答“流中是否有一个元素能匹配给定的谓词”。比如，你可以用它来看看菜单里面是否有素食可选择：

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
    System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

anyMatch方法返回一个boolean，因此是一个终端操作。

5.3.2　检查谓词是否匹配所有元素

allMatch方法的工作原理和anyMatch类似，但它会看看流中的元素是否都能匹配给定的谓

词。比如，你可以用它来看看菜品是否有利健康（即所有菜的热量都低于1000卡路里）：

```
boolean isHealthy = menu.stream()
                        .allMatch(d -> d.getCalories() < 1000);
```

noneMatch

和allMatch相对的是noneMatch。它可以确保流中没有任何元素与给定的谓词匹配。比如，你可以用noneMatch重写前面的例子：

```
boolean isHealthy = menu.stream()
                        .noneMatch(d -> d.getCalories() >= 1000);
```

anyMatch、allMatch和noneMatch这三个操作都用到了我们所谓的短路，这就是大家熟悉的Java中&&和||运算符短路在流中的版本。

短路求值

有些操作不需要处理整个流就能得到结果。例如，假设你需要对一个用and连起来的大布尔表达式求值。不管表达式有多长，你只需找到一个表达式为false，就可以推断整个表达式将返回false，所以用不着计算整个表达式。这就是**短路**。

对于流而言，某些操作（例如allMatch、anyMatch、noneMatch、findFirst和findAny）不用处理整个流就能得到结果。只要找到一个元素，就可以有结果了。同样，limit也是一个短路操作：它只需要创建一个给定大小的流，而用不着处理流中所有的元素。在碰到无限大小的流的时候，这种操作就有用了：它们可以把无限流变成有限流。我们会在5.7节中介绍无限流的例子。

5.3.3 查找元素

findAny方法将返回当前流中的任意元素。它可以与其他流操作结合使用。比如，你可能想找到一道素食菜肴。你可以结合使用filter和findAny方法来实现这个查询：

```
Optional<Dish> dish =
    menu.stream()
        .filter(Dish::isVegetarian)
        .findAny();
```

流水线将在后台进行优化使其只需走一遍，并在利用短路找到结果时立即结束。不过慢着，代码里面的Optional是个什么玩意儿？

Optional简介

Optional<T>类（java.util.Optional）是一个容器类，代表一个值存在或不存在。在上面的代码中，findAny可能什么元素都没找到。Java 8的库设计人员引入了Optional<T>，这样就不用返回众所周知容易出问题的null了。我们在这里不会详细讨论Optional，因为第10章会详细解释你的代码如何利用Optional，避免和null检查相关的bug。不过现在，了解一下

Optional里面几种可以迫使你显式地检查值是否存在或处理值不存在的情形的方法也不错。

- isPresent()将在Optional包含值的时候返回true, 否则返回false。
- ifPresent(Consumer<T> block)会在值存在的时候执行给定的代码块。我们在第3章介绍了Consumer函数式接口；它让你传递一个接收T类型参数，并返回void的Lambda表达式。
- T get()会在值存在时返回值，否则抛出一个NoSuchElement异常。
- T orElse(T other)会在值存在时返回值，否则返回一个默认值。

例如，在前面的代码中你需要显式地检查Optional对象中是否存在一道菜可以访问其名称：

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()                          <- 返回一个Optional<Dish>
    .ifPresent(d -> System.out.println(d.getName());   <- 如果包含一个值就打印它，否则什么都不做
```

5.3.4　查找第一个元素

有些流有一个出现顺序（encounter order）来指定流中项目出现的逻辑顺序（比如由List或排序好的数据列生成的流）。对于这种流，你可能想要找到第一个元素。为此有一个findFirst方法，它的工作方式类似于findany。例如，给定一个数字列表，下面的代码能找出第一个平方能被3整除的数：

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
    someNumbers.stream()
               .map(x -> x * x)
               .filter(x -> x % 3 == 0)
               .findFirst(); // 9
```

何时使用findFirst和findAny

你可能会想，为什么会同时有findFirst和findAny呢？答案是并行。找到第一个元素在并行上限制更多。如果你不关心返回的元素是哪个，请使用findAny，因为它在使用并行流时限制较少。

5.4　归约

到目前为止，你见到过的终端操作都是返回一个boolean（allMatch之类的）、void（forEach）或Optional对象（findAny等）。你也见过了使用collect来将流中的所有元素组合成一个List。

在本节中，你将看到如何把一个流中的元素组合起来，使用reduce操作来表达更复杂的查询，比如“计算菜单中的总卡路里”或“菜单中卡路里最高的菜是哪一个”。此类查询需要将流

中所有元素反复结合起来，得到一个值，比如一个Integer。这样的查询可以被归类为归约操作（将流归约成一个值）。用函数式编程语言的术语来说，这称为折叠（fold），因为你可以将这个操作看成把一张长长的纸（你的流）反复折叠成一个小方块，而这就是折叠操作的结果。

5.4.1 元素求和

在我们研究如何使用reduce方法之前，先来看看如何使用for-each循环来对数字列表中的元素求和：

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```


numbers中的每个元素都用加法运算符反复迭代来得到结果。通过反复使用加法，你把一个数字列表归约成了一个数字。这段代码中有两个参数：

- 总和变量的初始值，在这里是0；
- 将列表中所有元素结合在一起的操作，在这里是+。

要是还能把所有的数字相乘，而不必去复制粘贴这段代码，岂不是很好？这正是reduce操作的用武之地，它对这种重复应用的模式做了抽象。你可以像下面这样对流中所有的元素求和：

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

reduce接受两个参数：

- 一个初始值，这里是0；
- 一个BinaryOperator<T>来将两个元素结合起来产生一个新值，这里我们用的是lambda (a, b) -> a + b。

你也很容易把所有的元素相乘，只需要将另一个Lambda：(a, b) -> a * b传递给reduce操作就可以了：

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

图5-7展示了reduce操作是如何作用于一个流的：Lambda反复结合每个元素，直到流被归约成一个值。

让我们深入研究一下reduce操作是如何对一个数字流求和的。首先，0作为Lambda（a）的第一个参数，从流中获得4作为第二个参数（b）。0 + 4得到4，它成了新的累积值。然后再用累积值和流中下一个元素5调用Lambda，产生新的累积值9。接下来，再用累积值和下一个元素3调用Lambda，得到12。最后，用12和流中最后一个元素9调用Lambda，得到最终结果21。

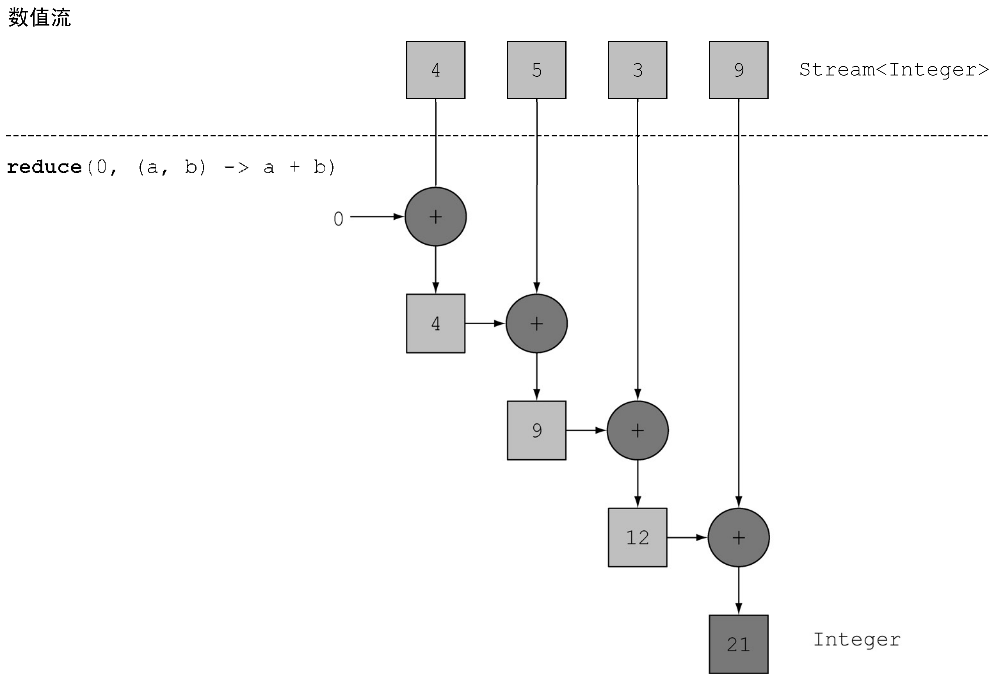


图5-7　使用reduce来对流中的数字求和

你可以使用方法引用让这段代码更简洁。在Java 8中，Integer类现在有了一个静态的sum方法来对两个数求和，这恰好是我们想要的，用不着反复用Lambda写同一段代码了：

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

无初始值

reduce还有一个重载的变体，它不接受初始值，但是会返回一个Optional对象：

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

为什么它返回一个Optional<Integer>呢？考虑流中没有任何元素的情况。reduce操作无法返回其和，因为它没有初始值。这就是为什么结果被包裹在一个Optional对象里，以表明和可能不存在。现在看看用reduce还能做什么。

5.4.2　最大值和最小值

原来，只要用归约就可以计算最大值和最小值了！让我们来看看如何利用刚刚学到的reduce来计算流中最大或最小的元素。正如你前面看到的，reduce接受两个参数：

- 一个初始值
- 一个Lambda来把两个流元素结合起来并产生一个新值

Lambda是一步步用加法运算符应用到流中每个元素上的，如图5-7所示。因此，你需要一个

给定两个元素能够返回最大值的Lambda。reduce操作会考虑新值和流中下一个元素，并产生一个新的最大值，直到整个流消耗完！你可以像下面这样使用reduce来计算流中的最大值，如图5-8所示。

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```

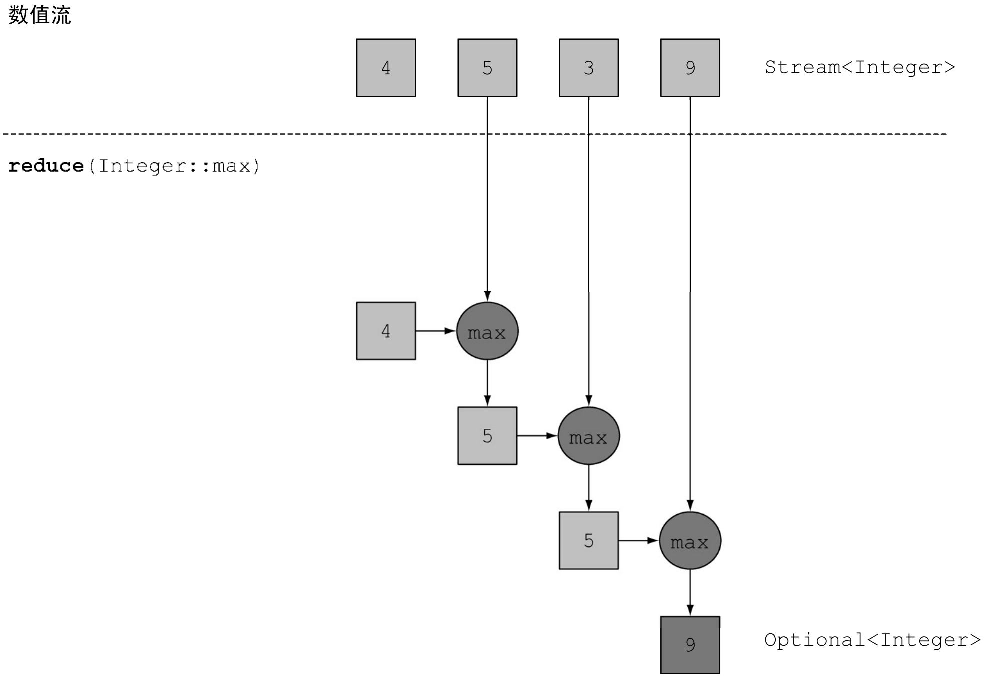


图5-8 一个归约操作——计算最大值

要计算最小值，你需要把Integer.min传给reduce来替换Integer.max：

```
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

你当然也可以写成Lambda (x, y) -> x < y ? x : y而不是Integer::min，不过后者比较易读。

为了检验你对于reduce操作的理解程度，试试测验5.3吧！

测验5.3：归约

怎样用map和reduce方法数一数流中有多少个菜呢？

答案：要解决这个问题，你可以把流中每个元素都映射成数字1，然后用reduce求和。这相当于按顺序数流中的元素个数。

```
int count = menu.stream()
                .map(d -> 1)
                .reduce(0, (a, b) -> a + b);
```

map和reduce的连接通常称为map-reduce模式，因Google用它来进行网络搜索而出名，因为它很容易并行化。请注意，在第4章中我们也看到了内置count方法可用来计算流中元素的个数：

```
long count = menu.stream().count();
```

归约方法的优势与并行化

相比于前面写的逐步迭代求和，使用reduce的好处在于，这里的迭代被内部迭代抽象掉了，这让内部实现得以选择并行执行reduce操作。而迭代式求和例子要更新共享变量sum，这不是那么容易并行化的。如果你加入了同步，很可能会发现线程竞争抵消了并行本应带来的性能提升！这种计算的并行化需要另一种办法：将输入分块，分块求和，最后再合并起来。但这样的话代码看起来就完全不一样了。你在第7章会看到使用分支/合并框架来做是什么样子。但现在重要的是要认识到，可变的累加器模式对于并行化来说是死路一条。你需要一种新的模式，这正是reduce所提供的。你还将在第7章看到，使用流来对所有的元素并行求和时，你的代码几乎不用修改：stream()换成了parallelStream()。

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

但要并行执行这段代码也要付一定代价，我们稍后会向你解释：传递给reduce的Lambda不能更改状态（如实例变量），而且操作必须满足结合律才可以按任意顺序执行。

到目前为止，你看到了产生一个Integer的归约例子：对流求和、流中的最大值，或是流中元素的个数。你将会在5.6节看到，诸如sum和max等内置的方法可以让常见归约模式的代码再简洁一点儿。我们会在下一章中讨论一种复杂的使用collect方法的归约。例如，如果你想要按类型对菜肴分组，也可以把流归约成一个Map而不是Integer。

流操作：无状态和有状态

你已经看到了很多的流操作。乍一看流操作简直是灵丹妙药，而且只要在从集合生成流的时候把Stream换成parallelStream就可以实现并行。

当然，对于许多应用来说确实是这样，就像前面的那些例子。你可以把一张菜单变成流，用filter选出某一类的菜肴，然后对得到的流做map来对卡路里求和，最后reduce得到菜单的总热量。这个流计算甚至可以并行进行。但这些操作的特性并不相同。它们需要操作的内部状态还是有些问题的。

诸如map或filter等操作会从输入流中获取每一个元素，并在输出流中得到0或1个结果。这些操作一般都是**无状态的**：它们没有内部状态（假设用户提供的Lambda或方法引用没有内部可变状态）。

但诸如reduce、sum、max等操作需要内部状态来累积结果。在上面的情况下，内部状态很小。在我们的例子里就是一个int或double。不管流中有多少元素要处理，内部状态都是

有界的。

相反，诸如sort或distinct等操作一开始都和filter和map差不多——都是接受一个流，再生成一个流（中间操作），但有一个关键的区别。从流中排序和删除重复项时都需要知道先前的历史。例如，排序要求所有元素都放入缓冲区后才能给输出流加入一个项目，这一操作的存储要求是无界的。要是流比较大或是无限的，就可能会有问题（把质数流倒序会做什么呢？它应当返回最大的质数，但数学告诉我们它不存在）。我们把这些操作叫作**有状态操作**。

你现在已经看到了很多流操作，可以用来表达复杂的数据处理查询。表5-1总结了迄今讲过的操作。你可以在下一节中通过一个练习来实践一下。

表5-1　中间操作和终端操作

操　作	类　型	返回类型	使用的类型/函数式接口	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
distinct	中间 (有状态-无界)	Stream<T>		
skip	中间 (有状态-有界)	Stream<T>	long	
limit	中间 (有状态-有界)	Stream<T>	long	
map	中间	Stream<R>	Function<T, R>	T -> R
flatMap	中间	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	中间 (有状态-无界)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	终端	boolean	Predicate<T>	T -> boolean
noneMatch	终端	boolean	Predicate<T>	T -> boolean
allMatch	终端	boolean	Predicate<T>	T -> boolean
findAny	终端	Optional<T>		
findFirst	终端	Optional<T>		
forEach	终端	void	Consumer<T>	T -> void
collect	终端	R	Collector<T, A, R>	
reduce	终端 (有状态-有界)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	终端	long		

5.5　付诸实践

在本节中，你会将迄今学到的关于流的知识付诸实践。我们来看一个不同的领域：执行交易

的交易员。你的经理让你为八个查询找到答案。你能做到吗？我们在5.5.2节给出了答案，但你应该自己先尝试一下作为练习。

(1) 找出2011年发生的所有交易，并按交易额排序（从低到高）。

(2) 交易员都在哪些不同的城市工作过？

(3) 查找所有来自于剑桥的交易员，并按姓名排序。

(4) 返回所有交易员的姓名字符串，按字母顺序排序。

(5) 有没有交易员是在米兰工作的？

(6) 打印生活在剑桥的交易员的所有交易额。

(7) 所有交易中，最高的交易额是多少？

(8) 找到交易额最小的交易。

5.5.1　领域：交易员和交易

以下是你要处理的领域，一个Traders和Transactions的列表：

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");

List<Transaction> transactions = Arrays.asList(
    new Transaction(brian, 2011, 300),
    new Transaction(raoul, 2012, 1000),
    new Transaction(raoul, 2011, 400),
    new Transaction(mario, 2012, 710),
    new Transaction(mario, 2012, 700),
    new Transaction(alan, 2012, 950)
);
```

Trader和Transaction类的定义如下：

```
public class Trader{

    private final String name;
    private final String city;

    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }

    public String getName(){
        return this.name;
    }

    public String getCity(){
        return this.city;
    }
```

```
    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
    }
}

public class Transaction{
    private final Trader trader;
    private final int year;
    private final int value;

    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }

    public Trader getTrader(){
        return this.trader;
    }

    public int getYear(){
        return this.year;
    }

    public int getValue(){
        return this.value;
    }

    public String toString(){
        return "{" + this.trader + ", " +
               "year: "+this.year+", " +
               "value:" + this.value +"}";
    }
}
```


5.5.2 解答

解答在下面的代码清单中。你可以看看你对迄今所学知识的理解程度如何。干得不错！

代码清单5-1 找出2011年的所有交易并按交易额排序（从低到高）

```
List<Transaction> tr2011 =
    transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());
```

- 给filter传递一个谓词来选择2011年的交易
- 按照交易额进行排序
- 将生成的Stream中的所有元素收集到一个List中

代码清单5-2 交易员都在哪些不同的城市工作过

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .distinct()
                .collect(toList());
```

- 提取与交易相关的每位交易员的所在城市
- 只选择互不相同的城市

这里还有一个新招：你可以去掉distinct()，改用toSet()，这样就会把流转换为集合。你在第6章中会了解到更多相关内容。

```
Set<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .collect(toSet());
```

代码清单5-3　查找所有来自于剑桥的交易员，并按姓名排序

```
List<Trader> traders =
    transactions.stream()
                .map(Transaction::getTrader)                     ← 从交易中提取所有交易员
                .filter(trader -> trader.getCity().equals("Cambridge"))   ← 仅选择位于剑桥的交易员
                .distinct()                                      ← 确保没有任何重复
                .sorted(comparing(Trader::getName))              ← 对生成的交易员流按照姓名进行排序
                .collect(toList());
```

代码清单5-4　返回所有交易员的姓名字符串，按字母顺序排序

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())   ← 提取所有交易员姓名，生成一个Strings构成的Stream
                .distinct()                                      ← 只选择不相同的姓名
                .sorted()                                        ← 对姓名按字母顺序排序
                .reduce("", (n1, n2) -> n1 + n2);                ← 逐个拼接每个名字，得到一个将所有名字连接起来的String
```

请注意，此解决方案效率不高（所有字符串都被反复连接，每次迭代的时候都要建立一个新的String对象）。下一章中，你将看到一个更为高效的解决方案，它像下面这样使用joining（其内部会用到StringBuilder）：

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

代码清单5-5　有没有交易员是在米兰工作的

```
boolean milanBased =
    transactions.stream()
                .anyMatch(transaction -> transaction.getTrader()
                                                    .getCity()
                                                    .equals("Milan"));
```
← 把一个谓词传递给anyMatch，检查是否有交易员在米兰工作

代码清单5-6　打印生活在剑桥的交易员的所有交易额

```
transactions.stream()
            .filter(t -> "Cambridge".equals(t.getTrader().getCity()))   ← 选择住在剑桥的交易员所进行的交易
            .map(Transaction::getValue)                                 ← 提取这些交易员的交易额
            .forEach(System.out::println);                              ← 打印每个值
```

代码清单5-7　所有交易中，最高的交易额是多少

```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)      <—— 提取每项交易的交易额
                .reduce(Integer::max);           <—— 计算生成的流中的最大值
```

代码清单5-8　找到交易额最小的交易

```
Optional<Transaction> smallestTransaction =
    transactions.stream()
                .reduce((t1, t2) ->
                         t1.getValue() < t2.getValue() ? t1 : t2);     <—— 通过反复比较每个交易的交易额，找出最小的交易
```

你还可以做得更好。流支持min和max方法，它们可以接受一个Comparator作为参数，指定计算最小或最大值时要比较哪个键值：

5

```
Optional<Transaction> smallestTransaction =
    transactions.stream()
                .min(comparing(Transaction::getValue));
```

5.6　数值流

我们在前面看到了可以使用reduce方法计算流中元素的总和。例如，你可以像下面这样计算菜单的热量：

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

这段代码的问题是，它有一个暗含的装箱成本。每个Integer都必须拆箱成一个原始类型，再进行求和。要是可以直接像下面这样调用sum方法，岂不是更好？

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .sum();
```

但这是不可能的。问题在于map方法会生成一个Stream<T>。虽然流中的元素是Integer类型，但Streams接口没有定义sum方法。为什么没有呢？比方说，你只有一个像menu那样的Stream<Dish>，把各种菜加起来是没有任何意义的。但不要担心，Stream API还提供了*原始类型流特化*，专门支持处理数值流的方法。

5.6.1　原始类型流特化

Java 8引入了三个原始类型特化流接口来解决这个问题：IntStream、DoubleStream和LongStream，分别将流中的元素特化为int、long和double，从而避免了暗含的装箱成本。每个接口都带来了进行常用数值归约的新方法，比如对数值流求和的sum，找到最大元素的max。此外还有在必要时再把它们转换回对象流的方法。要记住的是，这些特化的原因并不在于流的复

杂性，而是装箱造成的复杂性——即类似int和Integer之间的效率差异。

1. 映射到数值流

将流转换为特化版本的常用方法是mapToInt、mapToDouble和mapToLong。这些方法和前面说的map方法的工作方式一样，只是它们返回的是一个特化流，而不是Stream<T>。例如，你可以像下面这样用mapToInt对menu中的卡路里求和：

```
int calories = menu.stream()                       <-- 返回一个 Stream<Dish>
                    .mapToInt(Dish::getCalories)   <-- 返回一个 IntStream
                    .sum();
```

这里，mapToInt会从每道菜中提取热量（用一个Integer表示），并返回一个IntStream（而不是一个Stream<Integer>）。然后你就可以调用IntStream接口中定义的sum方法，对卡路里求和了！请注意，如果流是空的，sum默认返回0。IntStream还支持其他的方便方法，如max、min、average等。

2. 转换回对象流

同样，一旦有了数值流，你可能会想把它转换回非特化流。例如，IntStream上的操作只能产生原始整数：IntStream的map操作接受的Lambda必须接受int并返回int（一个IntUnaryOperator）。但是你可能想要生成另一类值，比如Dish。为此，你需要访问Stream接口中定义的那些更广义的操作。要把原始流转换成一般流（每个int都会装箱成一个Integer），可以使用boxed方法，如下所示：

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);   <-- 将Stream转换为数值流
Stream<Integer> stream = intStream.boxed();   <-- 将数值流转换为Stream
```

你在下一节中会看到，在需要将数值范围装箱成为一个一般流时，boxed尤其有用。

3. 默认值OptionalInt

求和的那个例子很容易，因为它有一个默认值：0。但是，如果你要计算IntStream中的最大元素，就得换个法子了，因为0是错误的结果。如何区分没有元素的流和最大值真的是0的流呢？前面我们介绍了Optional类，这是一个可以表示值存在或不存在的容器。Optional可以用Integer、String等参考类型来参数化。对于三种原始流特化，也分别有一个Optional原始类型特化版本：OptionalInt、OptionalDouble和OptionalLong。

例如，要找到IntStream中的最大元素，可以调用max方法，它会返回一个OptionalInt：

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

现在，如果没有最大值的话，你就可以显式处理OptionalInt去定义一个默认值了：

```
int max = maxCalories.orElse(1);   <-- 如果没有最大值的话，显式提供一个默认最大值
```

5.6.2　数值范围

和数字打交道时，有一个常用的东西就是数值范围。比如，假设你想要生成1和100之间的所

有数字。Java 8引入了两个可以用于IntStream和LongStream的静态方法，帮助生成这种范围：range和rangeClosed。这两个方法都是第一个参数接受起始值，第二个参数接受结束值。但range是不包含结束值的，而rangeClosed则包含结束值。让我们来看一个例子：

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                 .filter(n -> n % 2 == 0);

System.out.println(evenNumbers.count());
```

表示范围[1, 100]

一个从1到100的偶数流

从1到100有50个偶数

这里我们用了rangeClosed方法来生成1到100之间的所有数字。它会产生一个流，然后你可以链接filter方法，只选出偶数。到目前为止还没有进行任何计算。最后，你对生成的流调用count。因为count是一个终端操作，所以它会处理流，并返回结果50，这正是1到100（包括两端）中所有偶数的个数。请注意，比较一下，如果改用IntStream.range(1, 100)，则结果将会是49个偶数，因为range是不包含结束值的。

5

5.6.3　数值流应用：勾股数

现在我们来看一个难一点儿的例子，让你巩固一下有关数值流以及到目前为止学过的所有流操作的知识。如果你接受这个挑战，任务就是创建一个勾股数流。

1. 勾股数

那么什么是勾股数（毕达哥拉斯三元数）呢？我们得回到从前。在一堂激动人心的数学课上，你了解到，古希腊数学家毕达哥拉斯发现了某些三元数(a, b, c)满足公式a * a + b * b = c * c，其中a、b、c都是整数。例如，(3, 4, 5)就是一组有效的勾股数，因为3 * 3 + 4 * 4 = 5 * 5或9 + 16 = 25。这样的三元数有无限组。例如，(5, 12, 13)、(6, 8, 10)和(7, 24, 25)都是有效的勾股数。勾股数很有用，因为它们描述的正好是直角三角形的三条边长，如图5-9所示。

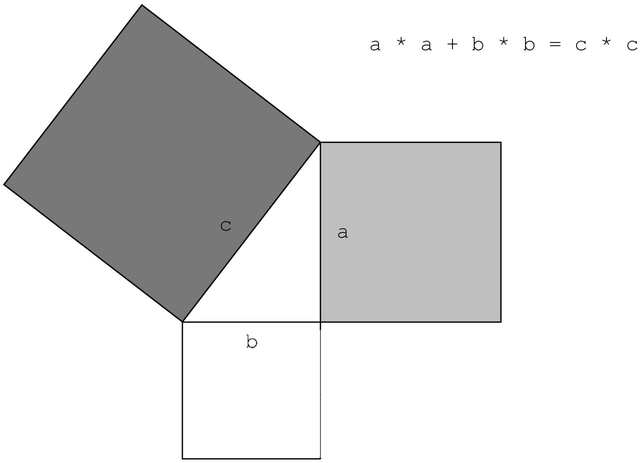


图5-9　勾股定理（毕达哥拉斯定理）

2. 表示三元数

那么，怎么入手呢？第一步是定义一个三元数。虽然更恰当的做法是定义一个新的类来表示三元数，但这里你可以使用具有三个元素的int数组，比如new int[]{3, 4, 5}，来表示勾股数(3, 4, 5)。现在你就可以用数组索引访问每个元素了。

3. 筛选成立的组合

假定有人为你提供了三元数中的前两个数字：a和b。怎么知道它是否能形成一组勾股数呢？你需要测试a * a + b * b的平方根是不是整数，也就是说它没有小数部分——在Java里可以使用expr % 1表示。如果它不是整数，那就是说c不是整数。你可以用filter操作表达这个要求（你稍后会了解到如何将其连接起来成为有效代码）：

```
filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
```

假设周围的代码给a提供了一个值，并且stream提供了b可能出现的值，filter将只选出那些可以与a组成勾股数的b。你可能在想Math.sqrt(a * a + b * b) % 1 == 0这一行是怎么回事。简单来说，这是一种测试Math.sqrt(a * a + b * b)返回的结果是不是整数的方法。如果平方根的结果带了小数，如9.1，这个条件就不成立（9.0是可以的）。

4. 生成三元组

在筛选之后，你知道a和b能够组成一个正确的组合。现在需要创建一个三元组。你可以使用map操作，像下面这样把每个元素转换成一个勾股数组：

```
stream.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
      .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

5. 生成b值

胜利在望！现在你需要生成b的值。前面已经看到，Stream.rangeClosed让你可以在给定区间内生成一个数值流。你可以用它来给b提供数值，这里是1到100：

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .boxed()
         .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

请注意，你在filter之后调用boxed，从rangeClosed返回的IntStream生成一个Stream<Integer>。这是因为你的map会为流中的每个元素返回一个int数组。而IntStream中的map方法只能为流中的每个元素返回另一个int，这可不是你想要的！你可以用IntStream的mapToObj方法改写它，这个方法会返回一个对象值流：

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

6. 生成值

这里有一个关键的假设：给出了a的值。 现在，只要已知a的值，你就有了一个可以生成勾股数的流。如何解决这个问题呢？就像b一样，你需要为a生成数值！最终的解决方案如下所示：

```
Stream<int[]> pythagoreanTriples =
    IntStream.rangeClosed(1, 100).boxed()
             .flatMap(a ->
                 IntStream.rangeClosed(a, 100)
                          .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
                          .mapToObj(b ->
                              new int[]{a, b, (int)Math.sqrt(a * a + b * b)})
                      );
```

好的，flatMap又是怎么回事呢？首先，创建一个从1到100的数值范围来生成a的值。对每个给定的a值，创建一个三元数流。要是把a的值映射到三元数流的话，就会得到一个由流构成的流。flatMap方法在做映射的同时，还会把所有生成的三元数流扁平化成一个流。这样你就得到了一个三元数流。还要注意，我们把b的范围改成了a到100。没有必要再从1开始了，否则就会造成重复的三元数，例如(3,4,5)和(4,3,5)。

7. 运行代码

现在你可以运行解决方案，并且可以利用我们前面看到的limit命令，明确限定从生成的流中要返回多少组勾股数了：

```
pythagoreanTriples.limit(5)
                  .forEach(t ->
                      System.out.println(t[0] + ", " + t[1] + ", " + t[2]));
```

这会打印：

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
```

8. 你还能做得更好吗？

目前的解决办法并不是最优的，因为你要求两次平方根。让代码更为紧凑的一种可能的方法是，先生成所有的三元数(a*a, b*b, a*a+b*b)，然后再筛选符合条件的：

```
Stream<double[]> pythagoreanTriples2 =
    IntStream.rangeClosed(1, 100).boxed()
             .flatMap(a ->
                 IntStream.rangeClosed(a, 100)
                          .mapToObj(
                              b -> new double[]{a, b, Math.sqrt(a*a + b*b)})   <- 产生三元数
                          .filter(t -> t[2] % 1 == 0));   <- 元组中的第三个元素必须是整数
```

5.7 构建流

希望到现在，我们已经让你相信，流对于表达数据处理查询是非常强大而有用的。到目前为止，你已经能够使用stream方法从集合生成流了。此外，我们还介绍了如何根据数值范围创建数值流。但创建流的方法还有许多！本节将介绍如何从值序列、数组、文件来创建流，甚至由生

成函数来创建无限流！

5.7.1　由值创建流

你可以使用静态方法Stream.of，通过显式值创建一个流。它可以接受任意数量的参数。例如，以下代码直接使用Stream.of创建了一个字符串流。然后，你可以将字符串转换为大写，再一个个打印出来：

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

你可以使用empty得到一个空流，如下所示：

```
Stream<String> emptyStream = Stream.empty();
```

5.7.2　由数组创建流

你可以使用静态方法Arrays.stream从数组创建一个流。它接受一个数组作为参数。例如，你可以将一个原始类型int的数组转换成一个IntStream，如下所示：

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();    <-- 总和是41
```

5.7.3　由文件生成流

Java中用于处理文件等I/O操作的NIO API（非阻塞 I/O）已更新，以便利用Stream API。java.nio.file.Files中的很多静态方法都会返回一个流。例如，一个很有用的方法是Files.lines，它会返回一个由指定文件中的各行构成的字符串流。使用你迄今所学的内容，你可以用这个方法看看一个文件中有多少各不相同的词：

```
long uniqueWords = 0;
try(Stream<String> lines =
          Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){    <-- 流会自动关闭
uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))    <-- 生成单词流
                   .distinct()    <-- 删除重复项
                   .count();    <-- 数一数有多少各不相同的单词
}
catch(IOException e){
                     <-- 如果打开文件时出现异常则加以处理
}
```

你可以使用Files.lines得到一个流，其中的每个元素都是给定文件中的一行。然后，你可以对line调用split方法将行拆分成单词。应该注意的是，你该如何使用flatMap产生一个扁平的单词流，而不是给每一行生成一个单词流。最后，把distinct和count方法链接起来，数数流中有多少各不相同的单词。

5.7.4 由函数生成流：创建无限流

Stream API提供了两个静态方法来从函数生成流：`Stream.iterate`和`Stream.generate`。这两个操作可以创建所谓的*无限流*：不像从固定集合创建的流那样有固定大小的流。由`iterate`和`generate`产生的流会用给定的函数按需创建值，因此可以无穷无尽地计算下去！一般来说，应该使用`limit(n)`来对这种流加以限制，以避免打印无穷多个值。

1. 迭代

我们先来看一个`iterate`的简单例子，然后再解释：

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```


`iterate`方法接受一个初始值（在这里是`0`），还有一个依次应用在每个产生的新值上的Lambda（`UnaryOperator<t>`类型）。这里，我们使用Lambda `n -> n + 2`，返回的是前一个元素加上2。因此，`iterate`方法生成了一个所有正偶数的流：流的第一个元素是初始值`0`。然后加上2来生成新的值`2`，再加上`2`来得到新的值`4`，以此类推。这种`iterate`操作基本上是顺序的，因为结果取决于前一次应用。请注意，此操作将生成一个*无限流*——这个流没有结尾，因为值是按需计算的，可以永远计算下去。我们说这个流是*无界*的。正如我们前面所讨论的，这是流和集合之间的一个关键区别。我们使用`limit`方法来显式限制流的大小。这里只选择了前10个偶数。然后可以调用`forEach`终端操作来消费流，并分别打印每个元素。

一般来说，在需要依次生成一系列值的时候应该使用`iterate`，比如一系列日期：1月31日，2月1日，依此类推。来看一个难一点儿的应用`iterate`的例子，试试测验5.4。

测验5.4：斐波纳契元组序列

斐波纳契数列是著名的经典编程练习。下面这个数列就是斐波纳契数列的一部分：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…数列中开始的两个数字是0和1，后续的每个数字都是前两个数字之和。

斐波纳契元组序列与此类似，是数列中数字和其后续数字组成的元组构成的序列：(0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21) …

你的任务是用`iterate`方法生成斐波纳契元组序列中的前20个元素。

让我们帮你入手吧。第一个问题是，`iterate`方法要接受一个`UnaryOperator<t>`作为参数，而你需要一个像(0,1)这样的元组流。你还是可以（这次又是比较草率地）使用一个数组的两个元素来代表元组。例如，`new int[]{0,1}`就代表了斐波纳契序列(0, 1)中的第一个元素。这就是`iterate`方法的初始值：

```
Stream.iterate(new int[]{0, 1}, ???)
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

在这个测验中，你需要搞清楚`???`代表的代码是什么。请记住，`iterate`会按顺序应用给定的Lambda。

答案：

```
Stream.iterate(new int[]{0, 1},
               t -> new int[]{t[1], t[0]+t[1]})
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

它是如何工作的呢？iterate需要一个Lambda来确定后续的元素。对于元组(3, 5)，其后续元素是(5, 3+5) = (5, 8)。下一个是(8, 5+8)。看到这个模式了吗？给定一个元组，其后续的元素是(t[1], t[0] + t[1])。这可以用这个Lambda来计算：t->new int[]{t[1], t[0]+t[1]}。运行这段代码，你就得到了序列(0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21)…请注意，如果你只想打印正常的斐波纳契数列，可以使用map提取每个元组中的第一个元素：

```
Stream.iterate(new int[]{0, 1},
               t -> new int[]{t[1],t[0] + t[1]})
      .limit(10)
      .map(t -> t[0])
      .forEach(System.out::println);
```

这段代码将生成斐波纳契数列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

2. 生成

与iterate方法类似，generate方法也可让你按需生成一个无限流。但generate不是依次对每个新生成的值应用函数的。它接受一个Supplier<T>类型的Lambda提供新的值。我们先来看一个简单的用法：

```
Stream.generate(Math::random)
      .limit(5)
      .forEach(System.out::println);
```

这段代码将生成一个流，其中有五个0到1之间的随机双精度数。例如，运行一次得到了下面的结果：

```
0.9410810294106129
0.6586270755634592
0.9592859117266873
0.13743396659487006
0.3942776037651241
```

Math.Random静态方法被用作新值生成器。同样，你可以用limit方法显式限制流的大小，否则流将会无限长。

你可能想知道，generate方法还有什么用途。我们使用的供应源（指向Math.random的方法引用）是无状态的：它不会在任何地方记录任何值，以备以后计算使用。但供应源不一定是无状态的。你可以创建存储状态的供应源，它可以修改状态，并在为流生成下一个值时使用。举个例子，我们将展示如何利用generate创建测验5.4中的斐波纳契数列，这样你就可以和用iterate方法的办法比较一下。但很重要的一点是，在并行代码中使用有状态的供应源是不安全的。因此下面的代码仅仅是为了内容完整，应尽量避免使用！我们会在第7章中进一步讨论这个

操作的问题和副作用，以及并行流。

我们在这个例子中会使用IntStream说明避免装箱操作的代码。IntStream的generate方法会接受一个IntSupplier，而不是Supplier<t>。例如，可以这样来生成一个全是1的无限流：

```
IntStream ones = IntStream.generate(() -> 1);
```

你在第3章中已经看到，Lambda允许你创建函数式接口的实例，只要直接内联提供方法的实现就可以。你也可以像下面这样，通过实现IntSupplier接口中定义的getAsInt方法显式传递一个对象（虽然这看起来是无缘无故地绕圈子，也请你耐心看）：

```
IntStream twos = IntStream.generate(new IntSupplier(){
            public int getAsInt(){
                return 2;
            }
        });
```


generate方法将使用给定的供应源，并反复调用getAsInt方法，而这个方法总是返回2。但这里使用的匿名类和Lambda的区别在于，匿名类可以通过字段定义状态，而状态又可以用getAsInt方法来修改。这是一个副作用的例子。你迄今见过的所有Lambda都是没有副作用的；它们没有改变任何状态。

回到斐波纳契数列的任务上，你现在需要做的是建立一个IntSupplier，它要把前一项的值保存在状态中，以便getAsInt用它来计算下一项。此外，在下一次调用它的时候，还要更新IntSupplier的状态。下面的代码就是如何创建一个在调用时返回下一个斐波纳契项的IntSupplier：

```
IntSupplier fib = new IntSupplier(){
    private int previous = 0;
    private int current = 1;
    public int getAsInt(){
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return oldPrevious;
    }
};
IntStream.generate(fib).limit(10).forEach(System.out::println);
```

前面的代码创建了一个IntSupplier的实例。此对象有可变的状态：它在两个实例变量中记录了前一个斐波纳契项和当前的斐波纳契项。getAsInt在调用时会改变对象的状态，由此在每次调用时产生新的值。相比之下，使用iterate的方法则是纯粹不变的：它没有修改现有状态，但在每次迭代时会创建新的元组。你将在第7章了解到，你应该始终采用不变的方法，以便并行处理流，并保持结果正确。请注意，因为你处理的是一个无限流，所以必须使用limit操作来显式限制它的大小；否则，终端操作（这里是forEach）将永远计算下去。同样，你不能对无限流做排序或归约，因为所有元素都需要处理，而这永远也完不成！

5.8　小结

这一章很长，但是很有收获！现在你可以更高效地处理集合了。事实上，流让你可以简洁地表达复杂的数据处理查询。此外，流可以透明地并行化。以下是你应从本章中学到的关键概念。

- Streams API可以表达复杂的数据处理查询。常用的流操作总结在表5-1中。
- 你可以使用`filter`、`distinct`、`skip`和`limit`对流做筛选和切片。
- 你可以使用`map`和`flatMap`提取或转换流中的元素。
- 你可以使用`findFirst`和`findAny`方法查找流中的元素。你可以用`allMatch`、`noneMatch`和`anyMatch`方法让流匹配给定的谓词。
- 这些方法都利用了短路：找到结果就立即停止计算；没有必要处理整个流。
- 你可以利用`reduce`方法将流中所有的元素迭代合并成一个结果，例如求和或查找最大元素。
- `filter`和`map`等操作是无状态的，它们并不存储任何状态。`reduce`等操作要存储状态才能计算出一个值。`sorted`和`distinct`等操作也要存储状态，因为它们需要把流中的所有元素缓存起来才能返回一个新的流。这种操作称为*有状态操作*。
- 流有三种基本的原始类型特化：`IntStream`、`DoubleStream`和`LongStream`。它们的操作也有相应的特化。
- 流不仅可以从集合创建，也可从值、数组、文件以及`iterate`与`generate`等特定方法创建。
- 无限流是没有固定大小的流。

第 6 章

用流收集数据

6

本章内容

- 用Collectors类创建和使用收集器
- 将数据流归约为一个值
- 汇总：归约的特殊情况
- 数据分组和分区
- 开发自己的自定义收集器

我们在前一章中学到，流可以用类似于数据库的操作帮助你处理集合。你可以把Java 8的流看作花哨又懒惰的数据集迭代器。它们支持两种类型的操作：中间操作（如filter或map）和终端操作（如count、findFirst、forEach和reduce）。中间操作可以链接起来，将一个流转换为另一个流。这些操作不会消耗流，其目的是建立一个流水线。与此相反，终端操作会消耗流，以产生一个最终结果，例如返回流中的最大元素。它们通常可以通过优化流水线来缩短计算时间。

我们已经在第4章和第5章中用过collect终端操作了，当时主要是用来把Stream中所有的元素结合成一个List。在本章中，你会发现collect是一个归约操作，就像reduce一样可以接受各种做法作为参数，将流中的元素累积成一个汇总结果。具体的做法是通过定义新的Collector接口来定义的，因此区分Collection、Collector和collect是很重要的。

下面是一些查询的例子，看看你用collect和收集器能够做什么。

- 对一个交易列表按货币分组，获得该货币的所有交易额总和（返回一个Map<Currency, Integer>）。
- 将交易列表分成两组：贵的和不贵的（返回一个Map<Boolean, List<Transaction>>）。
- 创建多级分组，比如按城市对交易分组，然后进一步按照贵或不贵分组（返回一个Map<Boolean, List<Transaction>>）。

激动吗？很好，我们先来看一个利用收集器的例子。想象一下，你有一个由Transaction构成的List，并且想按照名义货币进行分组。在没有Lambda的Java里，哪怕像这种简单的用例实现起来都很啰嗦，就像下面这样。

代码清单6-1　用指令式风格对交易按照货币分组

```
        Map<Currency, List<Transaction>> transactionsByCurrencies =
                                                  new HashMap<>();
  ┌──▷ for (Transaction transaction : transactions) {          ◁──迭代Transac-
  │  ┌─▷    Currency currency = transaction.getCurrency();         tion的List
建立累积交易分组的Map
  │  │     List<Transaction> transactionsForCurrency =
  │  │                       transactionsByCurrencies.get(currency);
  │  │     if (transactionsForCurrency == null) {        ◁──如果分组Map中没有这种货
  │  │         transactionsForCurrency = new ArrayList<>();  币的条目，就创建一个
提取Transaction的货币
  │  │         transactionsByCurrencies
  │  │                   .put(currency, transactionsForCurrency);
  │  │     }
  │  └     transactionsForCurrency.add(transaction);   ◁──将当前遍历的Transaction
       }                                                  加入同一货币的Transac-
                                                          tion的List
```

如果你是一位经验丰富的Java程序员，写这种东西可能挺顺手的，不过你必须承认，做这么简单的一件事就得写很多代码。更糟糕的是，读起来比写起来更费劲！代码的目的并不容易看出来，尽管换作白话的话是很直截了当的："把列表中的交易按货币分组。"你在本章中会学到，用Stream中collect方法的一个更通用的Collector参数，你就可以用一句话实现完全相同的结果，而用不着使用上一章中那个toList的特殊情况了：

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
        transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

这一比差得还真多，对吧？

6.1　收集器简介

前一个例子清楚地展示了函数式编程相对于指令式编程的一个主要优势：你只需指出希望的结果——"做什么"，而不用操心执行的步骤——"如何做"。在上一个例子里，传递给collect方法的参数是Collector接口的一个实现，也就是给Stream中元素做汇总的方法。上一章里的toList只是说"按顺序给每个元素生成一个列表"；在本例中，groupingBy说的是"生成一个Map，它的键是（货币）桶，值则是桶中那些元素的列表"。

要是做多级分组，指令式和函数式之间的区别就会更加明显：由于需要好多层嵌套循环和条件，指令式代码很快就变得更难阅读、更难维护、更难修改。相比之下，函数式版本只要再加上一个收集器就可以轻松地增强功能了，你会在6.3节中看到它。

6.1.1　收集器用作高级归约

刚刚的结论又引出了优秀的函数式API设计的另一个好处：更易复合和重用。收集器非常有用，因为用它可以简洁而灵活地定义collect用来生成结果集合的标准。更具体地说，对流调用collect方法将对流中的元素触发一个归约操作（由Collector来参数化）。图6-1所示的归约操作所做的工作和代码清单6-1中的指令式代码一样。它遍历流中的每个元素，并让Collector进

行处理。

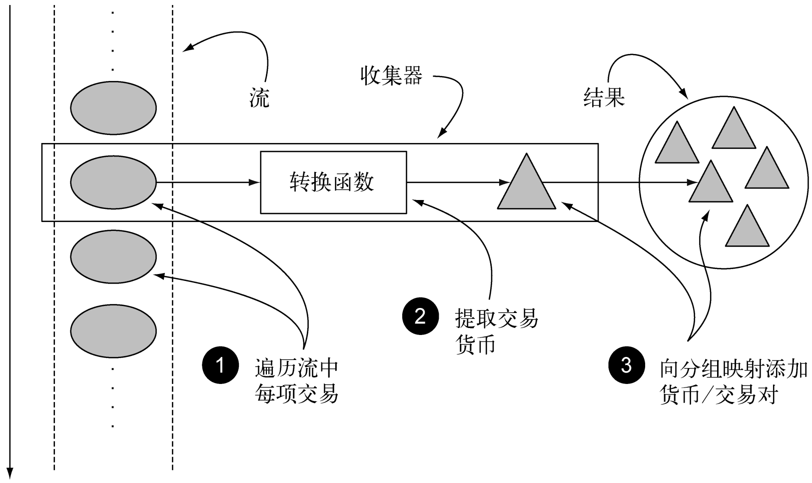


图6-1　按货币对交易分组的归约过程

一般来说，Collector会对元素应用一个转换函数（很多时候是不体现任何效果的恒等转换，例如toList），并将结果累积在一个数据结构中，从而产生这一过程的最终输出。例如，在前面所示的交易分组的例子中，转换函数提取了每笔交易的货币，随后使用货币作为键，将交易本身累积在生成的Map中。

如货币的例子中所示，Collector接口中方法的实现决定了如何对流执行归约操作。我们会在6.5节和6.6节研究如何创建自定义收集器。但Collectors实用类提供了很多静态工厂方法，可以方便地创建常见收集器的实例，只要拿来用就可以了。最直接和最常用的收集器是toList静态方法，它会把流中所有的元素收集到一个List中：

```
List<Transaction> transactions =
    transactionStream.collect(Collectors.toList());
```

6.1.2　预定义收集器

在本章剩下的部分中，我们主要探讨预定义收集器的功能，也就是那些可以从Collectors类提供的工厂方法（例如groupingBy）创建的收集器。它们主要提供了三大功能：

- 将流元素归约和汇总为一个值
- 元素分组
- 元素分区

我们先来看看可以进行归约和汇总的收集器。它们在很多场合下都很方便，比如前面例子中提到的求一系列交易的总交易额。

然后你将看到如何对流中的元素进行分组，同时把前一个例子推广到多层次分组，或把不同

的收集器结合起来，对每个子组进行进一步归约操作。我们还将谈到分组的特殊情况"分区"，即使用谓词（返回一个布尔值的单参数函数）作为分组函数。

6.4节末有一张表，总结了本章中探讨的所有预定义收集器。在6.5节你将了解更多有关Collector接口的内容。在6.6节中你会学到如何创建自己的自定义收集器，用于Collectors类的工厂方法无效的情况。

6.2 归约和汇总

为了说明从Collectors工厂类中能创建出多少种收集器实例，我们重用一下前一章的例子：包含一张佳肴列表的菜单！

就像你刚刚看到的，在需要将流项目重组成集合时，一般会使用收集器（Stream方法collect的参数）。再宽泛一点来说，但凡要把流中所有的项目合并成一个结果时就可以用。这个结果可以是任何类型，可以复杂如代表一棵树的多级映射，或是简单如一个整数——也许代表了菜单的热量总和。这两种结果类型我们都会讨论：6.2.2节讨论单个整数，6.3.1节讨论多级分组。

我们先来举一个简单的例子，利用counting工厂方法返回的收集器，数一数菜单里有多少种菜：

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

这还可以写得更为直接：

```
long howManyDishes = menu.stream().count();
```

counting收集器在和其他收集器联合使用的时候特别有用，后面会谈到这一点。

在本章后面的部分，我们假定你已导入了Collectors类的所有静态工厂方法：

```
import static java.util.stream.Collectors.*;
```

这样你就可以写counting()而用不着写Collectors.counting()之类的了。

让我们来继续探讨简单的预定义收集器，看看如何找到流中的最大值和最小值。

6.2.1 查找流中的最大值和最小值

假设你想要找出菜单中热量最高的菜。你可以使用两个收集器，Collectors.maxBy和Collectors.minBy，来计算流中的最大或最小值。这两个收集器接收一个Comparator参数来比较流中的元素。你可以创建一个Comparator来根据所含热量对菜肴进行比较，并把它传递给Collectors.maxBy：

```
Comparator<Dish> dishCaloriesComparator =
    Comparator.comparingInt(Dish::getCalories);

Optional<Dish> mostCalorieDish =
    menu.stream()
        .collect(maxBy(dishCaloriesComparator));
```

你可能在想Optional<Dish>是怎么回事。要回答这个问题，我们需要问“要是menu为空怎么办”。那就没有要返回的菜了！Java 8引入了Optional，它是一个容器，可以包含也可以不包含值。这里它完美地代表了可能也可能不返回菜肴的情况。我们在第5章讲findAny方法的时候简要提到过它。现在不用担心，我们专门用第10章来研究Optional<T>及其操作。

另一个常见的返回单个值的归约操作是对流中对象的一个数值字段求和。或者你可能想要求平均数。这种操作被称为汇总操作。让我们来看看如何使用收集器来表达汇总操作。

6.2.2　汇总

Collectors类专门为汇总提供了一个工厂方法：Collectors.summingInt。它可接受一个把对象映射为求和所需int的函数，并返回一个收集器；该收集器在传递给普通的collect方法后即执行我们需要的汇总操作。举个例子来说，你可以这样求出菜单列表的总热量：

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```


这里的收集过程如图6-2所示。在遍历流时，会把每一道菜都映射为其热量，然后把这个数字累加到一个累加器（这里的初始值0）。

Collectors.summingLong和Collectors.summingDouble方法的作用完全一样，可以用于求和字段为long或double的情况。

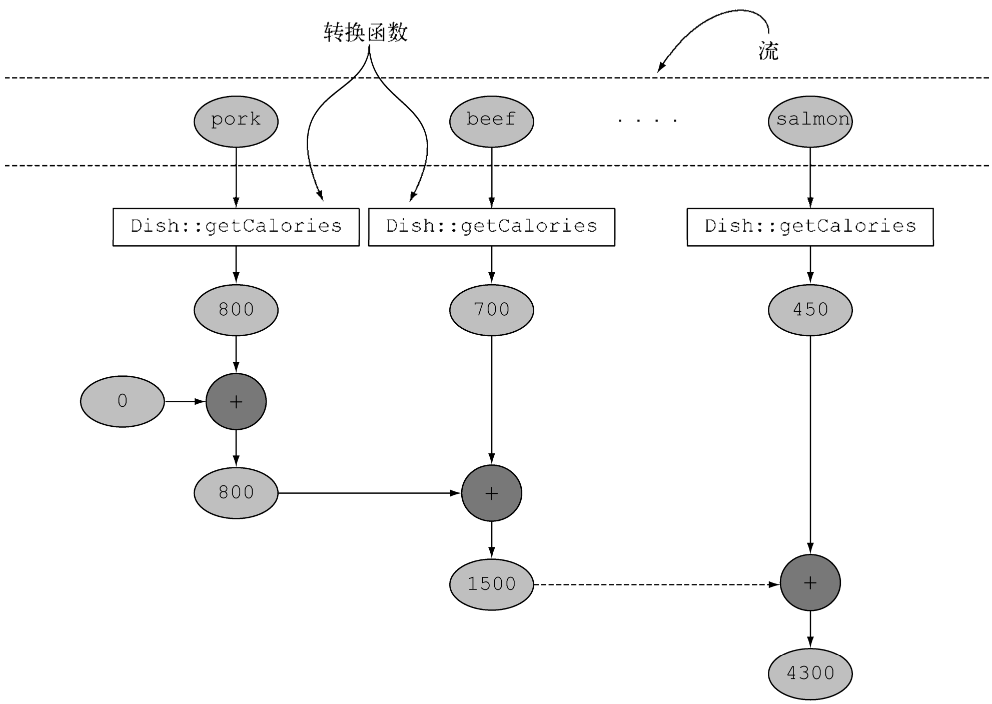


图6-2　summingInt收集器的累积过程

但汇总不仅仅是求和；还有Collectors.averagingInt，连同对应的averagingLong和averagingDouble可以计算数值的平均数：

```
double avgCalories =
    menu.stream().collect(averagingInt(Dish::getCalories));
```

到目前为止，你已经看到了如何使用收集器来给流中的元素计数，找到这些元素数值属性的最大值和最小值，以及计算其总和和平均值。不过很多时候，你可能想要得到两个或更多这样的结果，而且你希望只需一次操作就可以完成。在这种情况下，你可以使用summarizingInt工厂方法返回的收集器。例如，通过一次summarizing操作你可以就数出菜单中元素的个数，并得到菜肴热量总和、平均值、最大值和最小值：

```
IntSummaryStatistics menuStatistics =
        menu.stream().collect(summarizingInt(Dish::getCalories));
```

这个收集器会把所有这些信息收集到一个叫作IntSummaryStatistics的类里，它提供了方便的取值（getter）方法来访问结果。打印menuStatisticobject会得到以下输出：

```
IntSummaryStatistics{count=9, sum=4300, min=120,
                     average=477.777778, max=800}
```

同样，相应的summarizingLong和summarizingDouble工厂方法有相关的LongSummaryStatistics和DoubleSummaryStatistics类型，适用于收集的属性是原始类型long或double的情况。

6.2.3　连接字符串

joining工厂方法返回的收集器会把对流中每一个对象应用toString方法得到的所有字符串连接成一个字符串。这意味着你把菜单中所有菜肴的名称连接起来，如下所示：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

请注意，joining在内部使用了StringBuilder来把生成的字符串逐个追加起来。此外还要注意，如果Dish类有一个toString方法来返回菜肴的名称，那你无需用提取每一道菜名称的函数来对原流做映射就能够得到相同的结果：

```
String shortMenu = menu.stream().collect(joining());
```

二者均可产生以下字符串：

```
porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

但该字符串的可读性并不好。幸好，joining工厂方法有一个重载版本可以接受元素之间的分界符，这样你就可以得到一个逗号分隔的菜肴名称列表：

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

正如我们预期的那样，它会生成：

```
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

到目前为止，我们已经探讨了各种将流归约到一个值的收集器。在下一节中，我们会展示为什么所有这种形式的归约过程，其实都是Collectors.reducing工厂方法提供的更广义归约收集器的特殊情况。

6.2.4 广义的归约汇总

事实上，我们已经讨论的所有收集器，都是一个可以用reducing工厂方法定义的归约过程的特殊情况而已。Collectors.reducing工厂方法是所有这些特殊情况的一般化。可以说，先前讨论的案例仅仅是为了方便程序员而已。（但是，请记得方便程序员和可读性是头等大事！）例如，可以用reducing方法创建的收集器来计算你菜单的总热量，如下所示：

```
int totalCalories = menu.stream().collect(reducing(
                                   0, Dish::getCalories, (i, j) -> i + j));
```


它需要三个参数。

- 第一个参数是归约操作的起始值，也是流中没有元素时的返回值，所以很显然对于数值和而言0是一个合适的值。
- 第二个参数就是你在6.2.2节中使用的函数，将菜肴转换成一个表示其所含热量的int。
- 第三个参数是一个BinaryOperator，将两个项目累积成一个同类型的值。这里它就是对两个int求和。

同样，你可以使用下面这样单参数形式的reducing来找到热量最高的菜，如下所示：

```
Optional<Dish> mostCalorieDish =
    menu.stream().collect(reducing(
        (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

你可以把单参数reducing工厂方法创建的收集器看作三参数方法的特殊情况，它把流中的第一个项目作为起点，把*恒等函数*（即一个函数仅仅是返回其输入参数）作为一个转换函数。这也意味着，要是把单参数reducing收集器传递给空流的collect方法，收集器就没有起点；正如我们在6.2.1节中所解释的，它将因此而返回一个Optional<Dish>对象。

收集与归约

在上一章和本章中讨论了很多有关归约的内容。你可能想知道，Stream接口的collect和reduce方法有何不同，因为两种方法通常会获得相同的结果。例如，你可以像下面这样使用reduce方法来实现toListCollector所做的工作：

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> numbers = stream.reduce(
                               new ArrayList<Integer>(),
                               (List<Integer> l, Integer e) -> {
                                         l.add(e);
                                         return l; },
```

```
                                    (List<Integer> l1, List<Integer> l2) -> {
                                        l1.addAll(l2);
                                        return l1; });
```

这个解决方案有两个问题：一个语义问题和一个实际问题。语义问题在于，reduce方法旨在把两个值结合起来生成一个新值，它是一个不可变的归约。与此相反，collect方法的设计就是要改变容器，从而累积要输出的结果。这意味着，上面的代码片段是在滥用reduce方法，因为它在原地改变了作为累加器的List。你在下一章中会更详细地看到，以错误的语义使用reduce方法还会造成一个实际问题：这个归约过程不能并行工作，因为由多个线程并发修改同一个数据结构可能会破坏List本身。在这种情况下，如果你想要线程安全，就需要每次分配一个新的List，而对象分配又会影响性能。这就是collect方法特别适合表达可变容器上的归约的原因，更关键的是它适合并行操作，本章后面会谈到这一点。

1. 收集框架的灵活性：以不同的方法执行同样的操作

你还可以进一步简化前面使用reducing收集器的求和例子——引用Integer类的sum方法，而不用去写一个表达同一操作的Lambda表达式。这会得到以下程序：

```
int totalCalories = menu.stream().collect(reducing(0,     <-- 初始值
                                Dish::getCalories,      <-- 转换函数
                                Integer::sum));    <-- 累积函数
```

从逻辑上说，归约操作的工作原理如图6-3所示：利用累积函数，把一个初始化为起始值的累加器，和把转换函数应用到流中每个元素上得到的结果不断迭代合并起来。

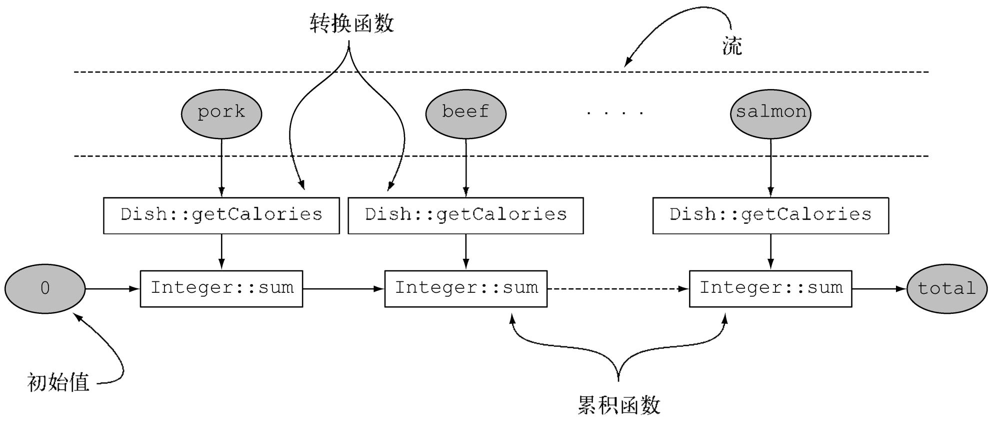


图6-3　计算菜单总热量的归约过程

在现实中，我们在6.2节开始时提到的counting收集器也是类似地利用三参数reducing工厂方法实现的。它把流中的每个元素都转换成一个值为1的Long型对象，然后再把它们相加：

```
public static <T> Collector<T, ?, Long> counting() {
    return reducing(0L, e -> 1L, Long::sum);
}
```

使用泛型?通配符

在刚刚提到的代码片段中，你可能已经注意到了?通配符，它用作counting工厂方法返回的收集器签名中的第二个泛型类型。对这种记法你应该已经很熟悉了，特别是如果你经常使用Java的集合框架的话。在这里，它仅仅意味着收集器的累加器类型未知，换句话说，累加器本身可以是任何类型。我们在这里原封不动地写出了Collectors类中原始定义的方法签名，但在本章其余部分我们将避免使用任何通配符表示法，以使讨论尽可能简单。

我们在第5章已经注意到，还有另一种方法不使用收集器也能执行相同操作——将菜肴流映射为每一道菜的热量，然后用前一个版本中使用的方法引用来归约得到的流：

```
int totalCalories =
    menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

请注意，就像流的任何单参数reduce操作一样，reduce(Integer::sum)返回的不是int而是Optional<Integer>，以便在空流的情况下安全地执行归约操作。然后你只需用Optional对象中的get方法来提取里面的值就行了。请注意，在这种情况下使用get方法是安全的，只是因为你已经确定菜肴流不为空。你在第10章还会进一步了解到，一般来说，使用允许提供默认值的方法，如orElse或orElseGet来解开Optional中包含的值更为安全。最后，更简洁的方法是把流映射到一个IntStream，然后调用sum方法，你也可以得到相同的结果：

```
int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();
```

2. 根据情况选择最佳解决方案

这再次说明了，函数式编程（特别是Java 8的Collections框架中加入的基于函数式风格原理设计的新API）通常提供了多种方法来执行同一个操作。这个例子还说明，收集器在某种程度上比Stream接口上直接提供的方法用起来更复杂，但好处在于它们能提供更高水平的抽象和概括，也更容易重用和自定义。

我们的建议是，尽可能为手头的问题探索不同的解决方案，但在通用的方案里面，始终选择最专门化的一个。无论是从可读性还是性能上看，这一般都是最好的决定。例如，要计菜单的总热量，我们更倾向于最后一个解决方案（使用IntStream），因为它最简明，也很可能最易读。同时，它也是性能最好的一个，因为IntStream可以让我们避免自动拆箱操作，也就是从Integer到int的隐式转换，它在这里毫无用处。

接下来，请看看测验6.1，测试一下你对于reducing作为其他收集器的概括的理解程度如何。

测验6.1：用reducing连接字符串

以下哪一种reducing收集器的用法能够合法地替代joining收集器（如6.2.3节用法）？

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

(1)
```
String shortMenu = menu.stream().map(Dish::getName)
                   .collect( reducing    ( (s1, s2) -> s1 + s2 ) ).get();
```

6

(2) `String shortMenu = menu.stream()`

```
.collect( reducing( (d1, d2) -> d1.getName() + d2.getName() ) ).get();
```

(3) `String shortMenu = menu.stream()`

```
.collect( reducing( "",Dish::getName, (s1, s2) -> s1 + s2 ) );
```

答案：语句1和语句3是有效的，语句2无法编译。

(1) 这会将每道菜转换为菜名，就像原先使用joining收集器的语句一样。然后用一个String作为累加器归约得到的字符串流，并将菜名逐个连接在它后面。

(2) 这无法编译，因为reducing接受的参数是一个BinaryOperator<t>，也就是一个BiFunction<T,T,T>。这就意味着它需要的函数必须能接受两个参数，然后返回一个相同类型的值，但这里用的Lambda表达式接受的参数是两个菜，返回的却是一个字符串。

(3) 这会把一个空字符串作为累加器来进行归约，在遍历菜肴流时，它会把每道菜转换成菜名，并追加到累加器上。请注意，我们前面讲过，reducing要返回一个Optional并不需要三个参数，因为如果是空流的话，它的返回值更有意义——也就是作为累加器初始值的空字符串。

请注意，虽然语句1和语句3都能够合法地替代joining收集器，它们在这里是用来展示我们为何可以（至少在概念上）把reducing看作本章中讨论的所有其他收集器的概括。然而就实际应用而言，不管是从可读性还是性能方面考虑，我们始终建议使用joining收集器。

6.3 分组

一个常见的数据库操作是根据一个或多个属性对集合中的项目进行分组。就像前面讲到按货币对交易进行分组的例子一样，如果用指令式风格来实现的话，这个操作可能会很麻烦、啰嗦而且容易出错。但是，如果用Java 8所推崇的函数式风格来重写的话，就很容易转化为一个非常容易看懂的语句。我们来看看这个功能的第二个例子：假设你要把菜单中的菜按照类型进行分类，有肉的放一组，有鱼的放一组，其他的都放另一组。用Collectors.groupingBy工厂方法返回的收集器就可以轻松地完成这项任务，如下所示：

```
Map<Dish.Type, List<Dish>> dishesByType =
                      menu.stream().collect(groupingBy(Dish::getType));
```

其结果是下面的Map：

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza],
MEAT=[pork, beef, chicken]}
```

这里，你给groupingBy方法传递了一个Function（以方法引用的形式），它提取了流中每一道Dish的Dish.Type。我们把这个Function叫作分类函数，因为它用来把流中的元素分成不同的组。如图6-4所示，分组操作的结果是一个Map，把分组函数返回的值作为映射的键，把流中所有具有这个分类值的项目的列表作为对应的映射值。在菜单分类的例子中，键就是菜的类型，

值就是包含所有对应类型的菜肴的列表。

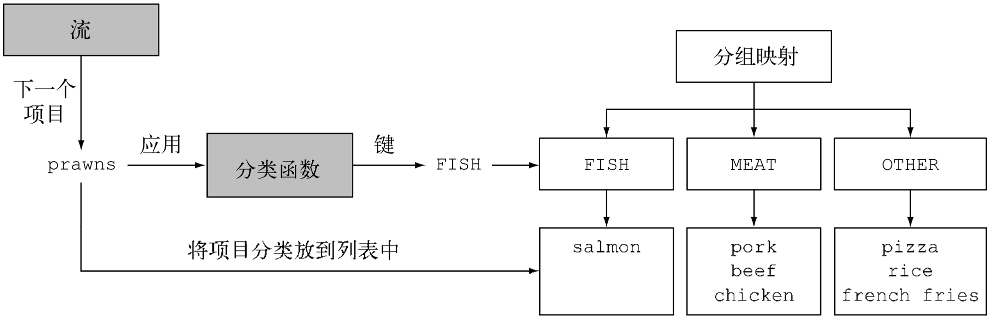


图6-4　在分组过程中对流中的项目进行分类

但是，分类函数不一定像方法引用那样可用，因为你想用以分类的条件可能比简单的属性访问器要复杂。例如，你可能想把热量不到400卡路里的菜划分为“低热量”（diet），热量400到700卡路里的菜划为“普通”（normal），高于700卡路里的划为“高热量”（fat）。由于`Dish`类的作者没有把这个操作写成一个方法，你无法使用方法引用，但你可以把这个逻辑写成Lambda表达式：

```
public enum CaloricLevel { DIET, NORMAL, FAT }

Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
        groupingBy(dish -> {
               if (dish.getCalories() <= 400) return CaloricLevel.DIET;
               else if (dish.getCalories() <= 700) return
    CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
         } ));
```

现在，你已经看到了如何对菜单中的菜肴按照类型和热量进行分组，但要是想同时按照这两个标准分类怎么办呢？分组的强大之处就在于它可以有效地组合。让我们来看看怎么做。

6.3.1　多级分组

要实现多级分组，我们可以使用一个由双参数版本的`Collectors.groupingBy`工厂方法创建的收集器，它除了普通的分类函数之外，还可以接受`collector`类型的第二个参数。那么要进行二级分组的话，我们可以把一个内层`groupingBy`传递给外层`groupingBy`，并定义一个为流中项目分类的二级标准，如代码清单6-2所示。

代码清单6-2　多级分组

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
menu.stream().collect(
      groupingBy(Dish::getType,
         groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
```

一级分类函数

二级分类函数

```
                else return CaloricLevel.FAT;
              } )
        )
);
```

这个二级分组的结果就是像下面这样的两级Map：

```
{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
 FISH={DIET=[prawns], NORMAL=[salmon]},
 OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}
```

这里的外层Map的键就是第一级分类函数生成的值：“fish, meat, other”，而这个Map的值又是一个Map，键是二级分类函数生成的值：“normal, diet, fat”。最后，第二级map的值是流中元素构成的List，是分别应用第一级和第二级分类函数所得到的对应第一级和第二级键的值：“salmon、pizza…” 这种多级分组操作可以扩展至任意层级，*n*级分组就会得到一个代表*n*级树形结构的*n*级Map。

图6-5显示了为什么结构相当于*n*维表格，并强调了分组操作的分类目的。

一般来说，把groupingBy看作“桶”比较容易明白。第一个groupingBy给每个键建立了一个桶。然后再用下游的收集器去收集每个桶中的元素，以此得到*n*级分组。

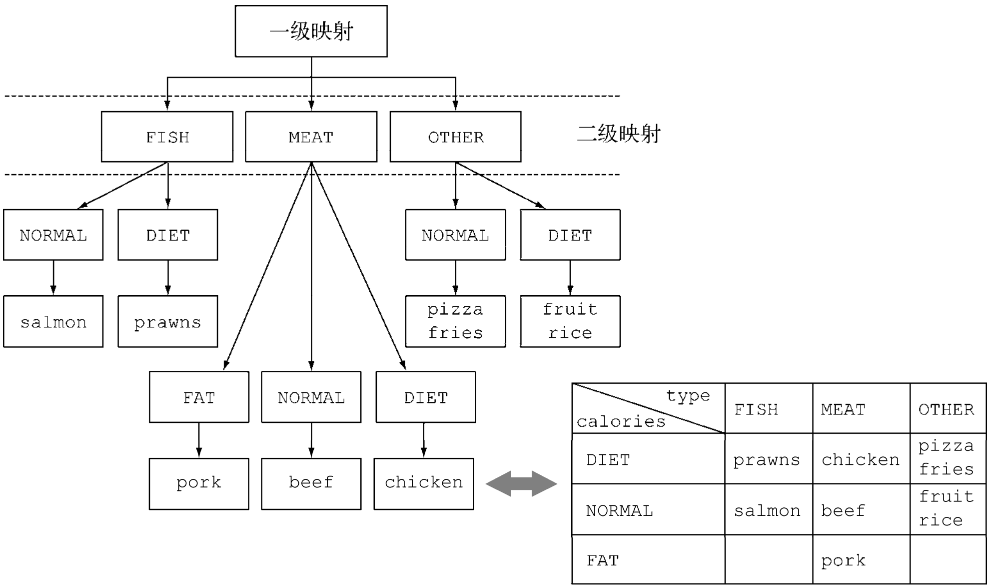


图6-5　*n*层嵌套映射和*n*维分类表之间的等价关系

6.3.2　按子组收集数据

在上一节中，我们看到可以把第二个groupingBy收集器传递给外层收集器来实现多级分组。但进一步说，传递给第一个groupingBy的第二个收集器可以是任何类型，而不一定是另一

个groupingBy。例如，要数一数菜单中每类菜有多少个，可以传递counting收集器作为groupingBy收集器的第二个参数：

```
Map<Dish.Type, Long> typesCount = menu.stream().collect(
                    groupingBy(Dish::getType, counting()));
```

其结果是下面的Map：

```
{MEAT=3, FISH=2, OTHER=4}
```

还要注意，普通的单参数groupingBy(f)（其中f是分类函数）实际上是groupingBy(f, toList())的简便写法。

再举一个例子，你可以把前面用于查找菜单中热量最高的菜肴的收集器改一改，按照菜的类型分类：

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                            maxBy(comparingInt(Dish::getCalories))));
```


这个分组的结果显然是一个map，以Dish的类型作为键，以包装了该类型中热量最高的Dish的Optional<Dish>作为值：

```
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}
```

注意　这个Map中的值是Optional，因为这是maxBy工厂方法生成的收集器的类型，但实际上，如果菜单中没有某一类型的Dish，这个类型就不会对应一个Optional. empty()值，而且根本不会出现在Map的键中。groupingBy收集器只有在应用分组条件后，第一次在流中找到某个键对应的元素时才会把键加入分组Map中。这意味着Optional包装器在这里不是很有用，因为它不会仅仅因为它是归约收集器的返回类型而表达一个最终可能不存在却意外存在的值。

1. 把收集器的结果转换为另一种类型

因为分组操作的Map结果中的每个值上包装的Optional没什么用，所以你可能想要把它们去掉。要做到这一点，或者更一般地来说，把收集器返回的结果转换为另一种类型，你可以使用Collectors.collectingAndThen工厂方法返回的收集器，如下所示。

代码清单6-3　查找每个子组中热量最高的Dish

```
Map<Dish.Type, Dish> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                 collectingAndThen(
                     maxBy(comparingInt(Dish::getCalories)),
                 Optional::get)));
```

分类函数

包装后的收集器

转换函数

这个工厂方法接受两个参数——要转换的收集器以及转换函数，并返回另一个收集器。这个收集器相当于旧收集器的一个包装，collect操作的最后一步就是将返回值用转换函数做一个映射。在这里，被包起来的收集器就是用maxBy建立的那个，而转换函数Optional::get则把返回的Optional中的值提取出来。前面已经说过，这个操作放在这里是安全的，因为reducing收集器永远都不会返回Optional.empty()。其结果是下面的Map：

```
{FISH=salmon, OTHER=pizza, MEAT=pork}
```

把好几个收集器嵌套起来很常见，它们之间到底发生了什么可能不那么明显。图6-6可以直观地展示它们是怎么工作的。从最外层开始逐层向里，注意以下几点。

- 收集器用虚线表示，因此groupingBy是最外层，根据菜肴的类型把菜单流分组，得到三个子流。
- groupingBy收集器包裹着collectingAndThen收集器，因此分组操作得到的每个子流都用这第二个收集器做进一步归约。
- collectingAndThen收集器又包裹着第三个收集器maxBy。
- 随后由归约收集器进行子流的归约操作，然后包含它的collectingAndThen收集器会对其结果应用Optional:get转换函数。
- 对三个子流分别执行这一过程并转换而得到的三个值，也就是各个类型中热量最高的Dish，将成为groupingBy收集器返回的Map中与各个分类键（Dish的类型）相关联的值。

2. 与groupingBy联合使用的其他收集器的例子

一般来说，通过groupingBy工厂方法的第二个参数传递的收集器将会对分到同一组中的所有流元素执行进一步归约操作。例如，你还重用求出所有菜肴热量总和的收集器，不过这次是对每一组Dish求和：

```
Map<Dish.Type, Integer> totalCaloriesByType =
               menu.stream().collect(groupingBy(Dish::getType,
                        summingInt(Dish::getCalories)));
```

然而常常和groupingBy联合使用的另一个收集器是mapping方法生成的。这个方法接受两个参数：一个函数对流中的元素做变换，另一个则将变换的结果对象收集起来。其目的是在累加之前对每个输入元素应用一个映射函数，这样就可以让接受特定类型元素的收集器适应不同类型的对象。我们来看一个使用这个收集器的实际例子。比方说你想要知道，对于每种类型的Dish，菜单中都有哪些CaloricLevel。我们可以把groupingBy和mapping收集器结合起来，如下所示：

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
   groupingBy(Dish::getType, mapping(
   dish -> { if (dish.getCalories() <= 400) return CaloricLevel.DIET;
           else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
       else return CaloricLevel.FAT; },
   toSet() )));
```

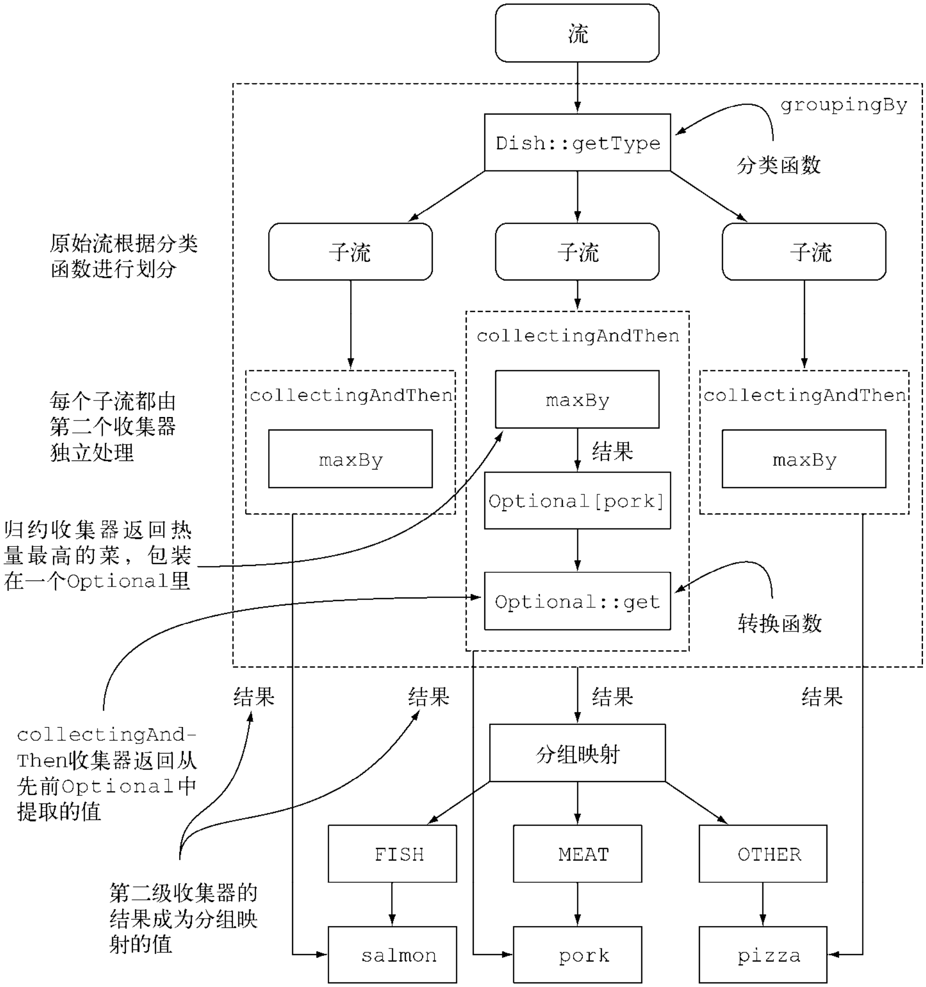


图6-6　嵌套收集器来获得多重效果

这里，就像我们前面见到过的，传递给映射方法的转换函数将Dish映射成了它的CaloricLevel：生成的CaloricLevel流传递给一个toSet收集器，它和toList类似，不过是把流中的元素累积到一个Set而不是List中，以便仅保留各不相同的值。如先前的示例所示，这个映射收集器将会收集分组函数生成的各个子流中的元素，让你得到这样的Map结果：

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, NORMAL, FAT], FISH=[DIET, NORMAL]}
```

由此你就可以轻松地做出选择了。如果你想吃鱼并且在减肥，那很容易找到一道菜；同样，如果你饥肠辘辘，想要很多热量的话，菜单中肉类部分就可以满足你的饕餮之欲了。请注意在上一个示例中，对于返回的Set是什么类型并没有任何保证。但通过使用toCollection，你就可以有更多的控制。例如，你可以给它传递一个构造函数引用来要求HashSet：

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
    groupingBy(Dish::getType, mapping(
    dish -> { if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT; },
    toCollection(HashSet::new) )));
```

6.4　分区

分区是分组的特殊情况：由一个谓词（返回一个布尔值的函数）作为分类函数，它称分区函数。分区函数返回一个布尔值，这意味着得到的分组Map的键类型是Boolean，于是它最多可以分为两组——true是一组，false是一组。例如，如果你是素食者或是请了一位素食的朋友来共进晚餐，可能会想要把菜单按照素食和非素食分开：

```
Map<Boolean, List<Dish>> partitionedMenu =
             menu.stream().collect(partitioningBy(Dish::isVegetarian));    <-- 分区函数
```

这会返回下面的Map：

```
{false=[pork, beef, chicken, prawns, salmon],
 true=[french fries, rice, season fruit, pizza]}
```

那么通过Map中键为true的值，就可以找出所有的素食菜肴了：

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

请注意，用同样的分区谓词，对菜单List创建的流作筛选，然后把结果收集到另外一个List中也可以获得相同的结果：

```
List<Dish> vegetarianDishes =
            menu.stream().filter(Dish::isVegetarian).collect(toList());
```

6.4.1　分区的优势

分区的好处在于保留了分区函数返回true或false的两套流元素列表。在上一个例子中，要得到非素食Dish的List，你可以使用两个筛选操作来访问partitionedMenu这个Map中false键的值：一个利用谓词，一个利用该谓词的非。而且就像你在分组中看到的，partitioningBy工厂方法有一个重载版本，可以像下面这样传递第二个收集器：

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
menu.stream().collect(
        partitioningBy(Dish::isVegetarian,                  <-- 分区函数
                       groupingBy(Dish::getType)));    <-- 第二个收集器
```

这将产生一个二级Map：

```
{false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},
```

```
true={OTHER=[french fries, rice, season fruit, pizza]}}
```

这里，对于分区产生的素食和非素食子流，分别按类型对菜肴分组，得到了一个二级Map，和6.3.1节的二级分组得到的结果类似。再举一个例子，你可以重用前面的代码来找到素食和非素食中热量最高的菜：

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
menu.stream().collect(
    partitioningBy(Dish::isVegetarian,
        collectingAndThen(
                          maxBy(comparingInt(Dish::getCalories)),
                          Optional::get)));
```

这将产生以下结果：

```
{false=pork, true=pizza}
```

我们在本节开始时说过，你可以把分区看作分组一种特殊情况。groupingBy和partitioningBy收集器之间的相似之处并不止于此；你在下一个测验中会看到，还可以按照和6.3.1节中分组类似的方式进行多级分区。

测验6.2：使用partitioningBy

我们已经看到，和groupingBy收集器类似，partitioningBy收集器也可以结合其他收集器使用。尤其是它可以与第二个partitioningBy收集器一起使用来实现多级分区。以下多级分区的结果会是什么呢？

(1) `menu.stream().collect(partitioningBy(Dish::isVegetarian,`
`partitioningBy (d -> d.getCalories() > 500)));`

(2) `menu.stream().collect(partitioningBy(Dish::isVegetarian,`
`partitioningBy (Dish::getType)));`

(3) `menu.stream().collect(partitioningBy(Dish::isVegetarian,`
`counting()));`

答案如下。

(1) 这是一个有效的多级分区，产生以下二级Map：

```
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},
  true={false=[rice, season fruit], true=[french fries, pizza]}}
```

(2) 这无法编译，因为partitioningBy需要一个谓词，也就是返回一个布尔值的函数。方法引用Dish::getType不能用作谓词。

(3) 它会计算每个分区中项目的数目，得到以下Map：

```
{false=5, true=4}
```

作为使用partitioningBy收集器的最后一个例子，我们把菜单数据模型放在一边，来看一个更为复杂也更为有趣的例子：将数字分为质数和非质数。

6.4.2　将数字按质数和非质数分区

假设你要写一个方法，它接受参数int *n*，并将前*n*个自然数分为质数和非质数。但首先，找出能够测试某一个待测数字是否是质数的谓词会很有帮助：

```
public boolean isPrime(int candidate) {
    return IntStream.range(2, candidate)
                    .noneMatch(i -> candidate % i == 0);
}
```

产生一个自然数范围，从2开始，直至但不包括待测数

如果待测数字不能被流中任何数字整除则返回true

一个简单的优化是仅测试小于等于待测数平方根的因子：

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

现在最主要的一部分工作已经做好了。为了把前*n*个数字分为质数和非质数，只要创建一个包含这*n*个数的流，用刚刚写的isPrime方法作为谓词，再给partitioningBy收集器归约就好了：

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                    .collect(
                        partitioningBy(candidate -> isPrime(candidate)));
}
```

现在我们已经讨论过了Collectors类的静态工厂方法能够创建的所有收集器，并介绍了使用它们的实际例子。表6-1将它们汇总到一起，给出了它们应用到Stream<T>上返回的类型，以及它们用于一个叫作menuStream的Stream<Dish>上的实际例子。

表6-1　Collectors类的静态工厂方法

工厂方法	返回类型	用　　于
toList	List<T>	把流中所有项目收集到一个 List
使用示例：List<Dish> dishes = menuStream.collect(toList());		
toSet	Set<T>	把流中所有项目收集到一个 Set，删除重复项
使用示例：Set<Dish> dishes = menuStream.collect(toSet());		
toCollection	Collection<T>	把流中所有项目收集到给定的供应源创建的集合
使用示例：Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);		
counting	Long	计算流中元素的个数
使用示例：long howManyDishes = menuStream.collect(counting());		
summingInt	Integer	对流中项目的一个整数属性求和

（续）

工厂方法	返回类型	用　　于
使用示例：int totalCalories = menuStream.collect(summingInt(Dish::getCalories));		
averagingInt	Double	计算流中项目 Integer 属性的平均值
使用示例：double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));		
summarizingInt	IntSummaryStatistics	收集关于流中项目 Integer 属性的统计值，例如最大、最小、总和与平均值
使用示例：IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));		
joining`	String	连接对流中每个项目调用 toString 方法所生成的字符串
使用示例：String shortMenu = menuStream.map(Dish::getName).collect(joining(", "));		
maxBy	Optional<T>	一个包裹了流中按照给定比较器选出的最大元素的 Optional，或如果流为空则为 Optional.empty()
使用示例：Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));		
minBy	Optional<T>	一个包裹了流中按照给定比较器选出的最小元素的 Optional，或如果流为空则为 Optional.empty()
使用示例：Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));		
reducing	归约操作产生的类型	从一个作为累加器的初始值开始，利用 BinaryOperator 与流中的元素逐个结合，从而将流归约为单个值
使用示例：int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));		
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果应用转换函数
使用示例：int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));		
groupingBy	Map<K, List<T>>	根据项目的一个属性的值对流中的项目作问组，并将属性值作为结果 Map 的键
使用示例：Map<Dish.Type,List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));		
partitioningBy	Map<Boolean,List<T>>	根据对流中每个项目应用谓词的结果来对项目进行分区
使用示例：Map<Boolean,List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));		

本章开头提到过，所有这些收集器都是对Collector接口的实现，因此我们会在本章剩余部分中详细讨论这个接口。我们会看看这个接口中的方法，然后探讨如何实现你自己的收集器。

6.5　收集器接口

Collector接口包含了一系列方法，为实现具体的归约操作（即收集器）提供了范本。我们

已经看过了Collector接口中实现的许多收集器，例如toList或groupingBy。这也意味着，你可以为Collector接口提供自己的实现，从而自由地创建自定义归约操作。在6.6节中，我们将展示如何实现Collector接口来创建一个收集器，来比先前更高效地将数值流划分为质数和非质数。

要开始使用Collector接口，我们先看看本章开始时讲到的一个收集器——toList工厂方法，它会把流中的所有元素收集成一个List。我们当时说在日常工作中经常会用到这个收集器，而且它也是写起来比较直观的一个，至少理论上如此。通过仔细研究这个收集器是怎么实现的，我们可以很好地了解Collector接口是怎么定义的，以及它的方法所返回的函数在内部是如何为collect方法所用的。

首先让我们在下面的列表中看看Collector接口的定义，它列出了接口的签名以及声明的五个方法。

代码清单6-4　Collector接口

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

本列表适用以下定义。

- T是流中要收集的项目的泛型。
- A是累加器的类型，累加器是在收集过程中用于累积部分结果的对象。
- R是收集操作得到的对象（通常但并不一定是集合）的类型。

例如，你可以实现一个ToListCollector<T>类，将Stream<T>中的所有元素收集到一个List<T>里，它的签名如下：

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

我们很快就会澄清，这里用于累积的对象也将是收集过程的最终结果。

6.5.1　理解 Collector 接口声明的方法

现在我们可以一个个来分析Collector接口声明的五个方法了。通过分析，你会注意到，前四个方法都会返回一个会被collect方法调用的函数，而第五个方法characteristics则提供了一系列特征，也就是一个提示列表，告诉collect方法在执行归约操作的时候可以应用哪些优化（比如并行化）。

1. 建立新的结果容器：supplier方法

supplier方法必须返回一个结果为空的Supplier，也就是一个无参数函数，在调用时它会创建一个空的累加器实例，供数据收集过程使用。很明显，对于将累加器本身作为结果返回的收集器，比如我们的ToListCollector，在对空流执行操作的时候，这个空的累加器也代表了收

集过程的结果。在我们的ToListCollector中，supplier返回一个空的List，如下所示：

```
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
}
```

请注意你也可以只传递一个构造函数引用：

```
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}
```

2. 将元素添加到结果容器：accumulator方法

accumulator方法会返回执行归约操作的函数。当遍历到流中第n个元素时，这个函数执行时会有两个参数：保存归约结果的累加器（已收集了流中的前 n–1 个项目），还有第n个元素本身。该函数将返回void，因为累加器是原位更新，即函数的执行改变了它的内部状态以体现遍历的元素的效果。对于ToListCollector，这个函数仅仅会把当前项目添加至已经遍历过的项目的列表：


```
public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
}
```

你也可以使用方法引用，这会更为简洁：

```
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

3. 对结果容器应用最终转换：finisher方法

在遍历完流后，finisher方法必须返回在累积过程的最后要调用的一个函数，以便将累加器对象转换为整个集合操作的最终结果。通常，就像ToListCollector的情况一样，累加器对象恰好符合预期的最终结果，因此无需进行转换。所以finisher方法只需返回identity函数：

```
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

这三个方法已经足以对流进行顺序归约，至少从逻辑上看可以按图6-7进行。实践中的实现细节可能还要复杂一点，一方面是因为流的延迟性质，可能在collect操作之前还需要完成其他中间操作的流水线，另一方面则是理论上可能要进行并行归约。

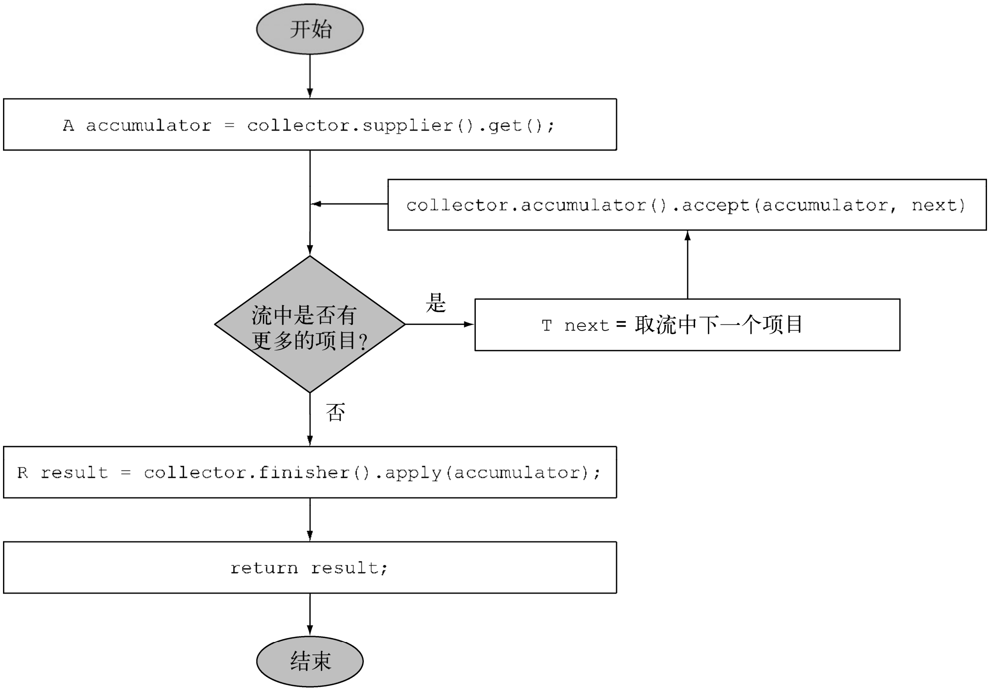


图6-7　顺序归约过程的逻辑步骤

4. 合并两个结果容器：combiner方法

四个方法中的最后一个——combiner方法会返回一个供归约操作使用的函数，它定义了对流的各个子部分进行并行处理时，各个子部分归约所得的累加器要如何合并。对于toList而言，这个方法的实现非常简单，只要把从流的第二个部分收集到的项目列表加到遍历第一部分时得到的列表后面就行了：

```
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1; }
}
```

有了这第四个方法，就可以对流进行并行归约了。它会用到Java 7中引入的分支/合并框架和Spliterator抽象，我们会在下一章中讲到。这个过程类似于图6-8所示，这里会详细介绍。

- 原始流会以递归方式拆分为子流，直到定义流是否需要进一步拆分的一个条件为非（如果分布式工作单位太小，并行计算往往比顺序计算要慢，而且要是生成的并行任务比处理器内核数多很多的话就毫无意义了）。
- 现在，所有的子流都可以并行处理，即对每个子流应用图6-7所示的顺序归约算法。
- 最后，使用收集器combiner方法返回的函数，将所有的部分结果两两合并。这时会把原始流每次拆分时得到的子流对应的结果合并起来。

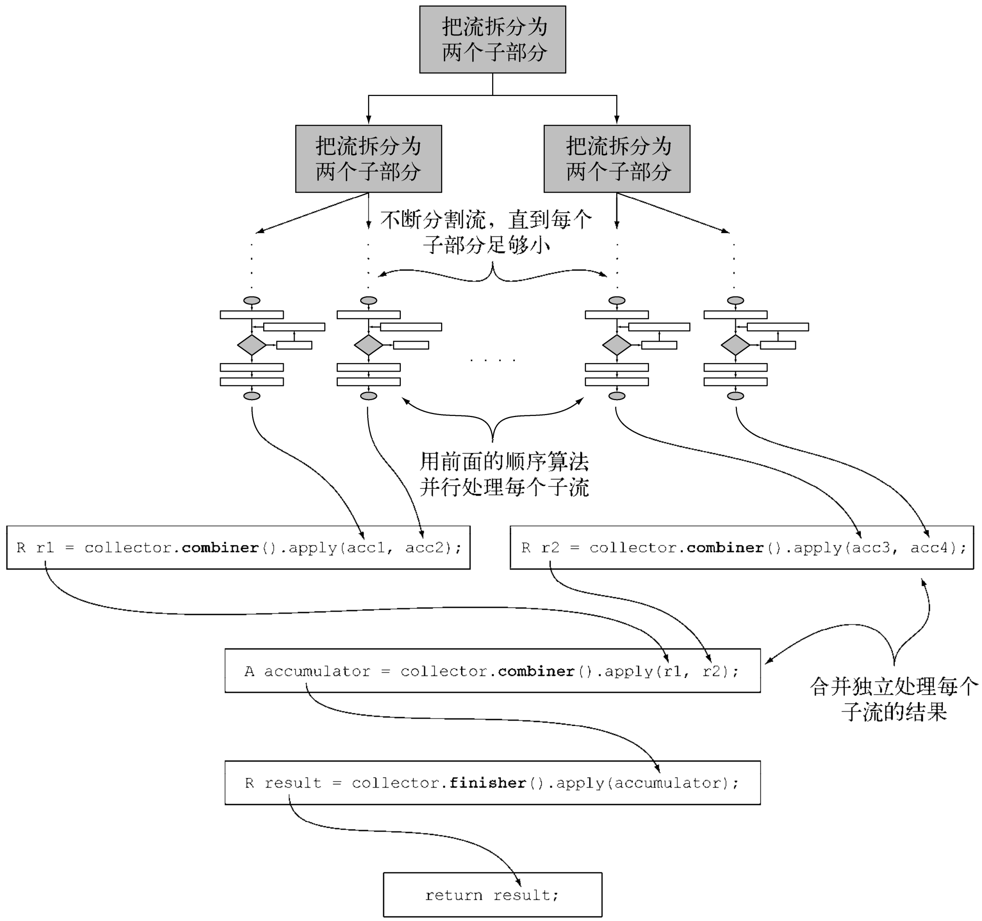


图6-8 使用combiner方法来并行化归约过程

5. characteristics方法

最后一个方法——characteristics会返回一个不可变的Characteristics集合，它定义了收集器的行为——尤其是关于流是否可以并行归约，以及可以使用哪些优化的提示。Characteristics是一个包含三个项目的枚举。

- UNORDERED——归约结果不受流中项目的遍历和累积顺序的影响。
- CONCURRENT——accumulator函数可以从多个线程同时调用，且该收集器可以并行归约流。如果收集器没有标为UNORDERED，那它仅在用于无序数据源时才可以并行归约。
- IDENTITY_FINISH——这表明完成器方法返回的函数是一个恒等函数，可以跳过。这种情况下，累加器对象将会直接用作归约过程的最终结果。这也意味着，将累加器A不加检查地转换为结果R是安全的。

我们迄今开发的ToListCollector是IDENTITY_FINISH的，因为用来累积流中元素的

List已经是我们要的最终结果，用不着进一步转换了，但它并不是UNORDERED，因为用在有序流上的时候，我们还是希望顺序能够保留在得到的List中。最后，它是CONCURRENT的，但我们刚才说过了，仅仅在背后的数据源无序时才会并行处理。

6.5.2　全部融合到一起

前一小节中谈到的五个方法足够我们开发自己的ToListCollector了。你可以把它们都融合起来，如下面的代码清单所示。

代码清单6-5　ToListCollector

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;

public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {

    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;                  <-- 创建集合操作的起始点
    }

    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;                       <-- 累积遍历过的项目，原位修改累加器
    }

    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.indentity();            <-- 恒等函数
    }

    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);                <-- 修改第一个累加器，将其与第二个累加器的内容合并
            return list1;                       <-- 返回修改后的第一个累加器
        };
    }

    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(
            IDENTITY_FINISH, CONCURRENT));      <-- 为收集器添加IDENTITY_FINISH和CONCURRENT标志
    }
}
```

请注意，这个实现与Collectors.toList方法并不完全相同，但区别仅仅是一些小的优化。这些优化的一个主要方面是Java API所提供的收集器在需要返回空列表时使用了Collections.

emptyList()这个单例（singleton）。这意味着它可安全地替代原生Java，来收集菜单流中的所有Dish的列表：

```
List<Dish> dishes = menuStream.collect(new ToListCollector<Dish>());
```

这个实现和标准的

```
List<Dish> dishes = menuStream.collect(toList());
```

构造之间的其他差异在于toList是一个工厂，而ToListCollector必须用new来实例化。

进行自定义收集而不去实现Collector

对于IDENTITY_FINISH的收集操作，还有一种方法可以得到同样的结果而无需从头实现新的Collectors接口。Stream有一个重载的collect方法可以接受另外三个函数——supplier、accumulator和combiner，其语义和Collector接口的相应方法返回的函数完全相同。所以比如说，我们可以像下面这样把菜肴流中的项目收集到一个List中：

```
List<Dish> dishes = menuStream.collect(
                        ArrayList::new,      <-- 供应源
                        List::add,           <-- 累加器
                        List::addAll);       <-- 组合器
```

我们认为，这第二种形式虽然比前一个写法更为紧凑和简洁，却不那么易读。此外，以恰当的类来实现自己的自定义收集器有助于重用并可避免代码重复。另外值得注意的是，这第二个collect方法不能传递任何Characteristics，所以它永远都是一个IDENTITY_FINISH和CONCURRENT但并非UNORDERED的收集器。

在下一节中，我们会让你实现收集器的新知识更上一层楼。你将会为一个更为复杂，但更为具体、更有说服力的用例开发自己的自定义收集器。

6.6　开发你自己的收集器以获得更好的性能

在6.4节讨论分区的时候，我们用Collectors类提供的一个方便的工厂方法创建了一个收集器，它将前*n*个自然数划分为质数和非质数，如下所示。

代码清单6-6　将前*n*个自然数按质数和非质数分区

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                    .collect(partitioningBy(candidate -> isPrime(candidate));
}
```

当时，通过限制除数不超过被测试数的平方根，我们对最初的isPrime方法做了一些改进：

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

还有没有办法来获得更好的性能呢？答案是“有”，但为此你必须开发一个自定义收集器。

6.6.1　仅用质数做除数

一个可能的优化是仅仅看看被测试数是不是能够被质数整除。要是除数本身都不是质数就用不着测了。所以我们可以仅仅用被测试数之前的质数来测试。然而我们目前所见的预定义收集器的问题，也就是必须自己开发一个收集器的原因在于，在收集过程中是没有办法访问部分结果的。这意味着，当测试某一个数字是否是质数的时候，你没法访问目前已经找到的其他质数的列表。

假设你有这个列表，那就可以把它传给isPrime方法，将方法重写如下：

```
public static boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}
```

而且还应该应用先前的优化，仅仅用小于被测数平方根的质数来测试。因此，你需要想办法在下一个质数大于被测数平方根时立即停止测试。不幸的是，Stream API中没有这样一种方法。你可以使用filter(p -> p <= candidateRoot)来筛选出小于被测数平方根的质数。但filter要处理整个流才能返回恰当的结果。如果质数和非质数的列表都非常大，这就是个问题了。你用不着这样做；你只需在质数大于被测数平方根的时候停下来就可以了。因此，我们会创建一个名为takeWhile的方法，给定一个排序列表和一个谓词，它会返回元素满足谓词的最长前缀：

```
public static <A> List<A> takeWhile(List<A> list, Predicate<A> p) {
    int i = 0;
    for (A item : list) {
        if (!p.test(item)) {                  <— 检查列表中的当前项目是否满足谓词
            return list.subList(0, i);        <— 如果不满足，返回该项目之前的前缀子列表
        }
        i++;
    }
    return list;      <— 列表中的所有项目都满足谓词，因此返回列表本身
}
```

利用这个方法，你就可以优化isPrime方法，只用不大于被测数平方根的质数去测试了：

```
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return takeWhile(primes, i -> i <= candidateRoot)
               .stream()
               .noneMatch(p -> candidate % p == 0);
}
```

请注意，这个takeWhile实现是即时的。理想情况下，我们会想要一个延迟求值的takeWhile，这样就可以和noneMatch操作合并。不幸的是，这样的实现超出了本章的范围，你需要了解Stream API的实现才行。

有了这个新的isPrime方法在手，你就可以实现自己的自定义收集器了。首先要声明一个实现Collector接口的新类，然后要开发Collector接口所需的五个方法。

1. 第一步：定义Collector类的签名

让我们从类签名开始吧，记得Collector接口的定义是：

```
public interface Collector<T, A, R>
```

其中T、A和R分别是流中元素的类型、用于累积部分结果的对象类型，以及collect操作最终结果的类型。这里应该收集Integer流，而累加器和结果类型则都是Map<Boolean, List<Integer>>（和先前代码清单6-6中分区操作得到的结果Map相同），键是true和false，值则分别是质数和非质数的List：

```
public class PrimeNumbersCollector
        implements Collector<Integer,                          ← 流中元素的类型
                             Map<Boolean, List<Integer>>,      ← 累加器类型
                             Map<Boolean, List<Integer>>>      ← collect操作的结果类型
```

2. 第二步：实现归约过程

接下来，你需要实现Collector接口中声明的五个方法。supplier方法会返回一个在调用时创建累加器的函数：

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {{
        put(true, new ArrayList<Integer>());
        put(false, new ArrayList<Integer>());
    }};
}
```

这里不但创建了用作累加器的Map，还为true和false两个键下面初始化了对应的空列表。在收集过程中会把质数和非质数分别添加到这里。收集器中最重要的方法是accumulator，因为它定义了如何收集流中元素的逻辑。这里它也是实现前面所讲的优化的关键。现在在任何一次迭代中，都可以访问收集过程的部分结果，也就是包含迄今找到的质数的累加器：

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime(acc.get(true), candidate) )   ← 根据isPrime的结果，获取质数或非质数列表
               .add(candidate);                            ← 将被测数添加到相应的列表中
    };
}
```

在这个方法中，你调用了isPrime方法，将待测试是否为质数的数以及迄今找到的质数列表（也就是累积Map中true键对应的值）传递给它。这次调用的结果随后被用作获取质数或非质数列表的键，这样就可以把新的被测数添加到恰当的列表中。

3. 第三步：让收集器并行工作（如果可能）

下一个方法要在并行收集时把两个部分累加器合并起来，这里，它只需要合并两个Map，即将第二个Map中质数和非质数列表中的所有数字合并到第一个Map的对应列表中就行了：

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1,
            Map<Boolean, List<Integer>> map2) -> {
```

```
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
                return map1;
        };
    }
```

请注意，实际上这个收集器是不能并行使用的，因为该算法本身是顺序的。这意味着永远都不会调用combiner方法，你可以把它的实现留空（更好的做法是抛出一个Unsupported-OperationException异常）。为了让这个例子完整，我们还是决定实现它。

4. 第四步：finisher方法和收集器的characteristics方法

最后两个方法的实现都很简单。前面说过，accumulator正好就是收集器的结果，用不着进一步转换，那么finisher方法就返回identity函数：

```
public Function<Map<Boolean, List<Integer>>,
                Map<Boolean, List<Integer>>> finisher() {
        return Function.identity();
}
```

就characteristics方法而言，我们已经说过，它既不是CONCURRENT也不是UNORDERED，但却是IDENTITY_FINISH的：

```
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
}
```

下面列出了最后实现的PrimeNumbersCollector。

代码清单6-7　PrimeNumbersCollector

```
public class PrimeNumbersCollector
    implements Collector<Integer,
                         Map<Boolean, List<Integer>>,
                         Map<Boolean, List<Integer>>> {
    @Override
    public Supplier<Map<Boolean, List<Integer>>> supplier() {
        return () -> new HashMap<Boolean, List<Integer>>() {{     <-- 从一个有两个空List的Map开始收集过程
            put(true, new ArrayList<Integer>());
            put(false, new ArrayList<Integer>());
        }};
    }

    @Override
    public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
        return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime( acc.get(true),      <-- 将已经找到的质数列表传递给isPrime方法
                              candidate) )
               .add(candidate);     <-- 根据isPrime方法的返回值，从Map中取质数或非质数列表，把当前的被测数加进去
        };
    }`

    @Override
    public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
```

```
        return (Map<Boolean, List<Integer>> map1,
                Map<Boolean, List<Integer>> map2) -> {
                    map1.get(true).addAll(map2.get(true));
                    map1.get(false).addAll(map2.get(false));
                    return map1;
            };
    }

    @Override
    public Function<Map<Boolean, List<Integer>>,
                    Map<Boolean, List<Integer>>> finisher() {
        return Function.identity();
    }

    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }
}
```

将第二个Map合并到第一个

收集过程最后无需转换，因此用identity函数收尾

这个收集器是IDENTITY_FINISH，但既不是UNORDERED也不是CONCURRENT，因为质数是按顺序发现的

现在你可以用这个新的自定义收集器来代替6.4节中用partitioningBy工厂方法创建的那个，并获得完全相同的结果了：

```
public Map<Boolean, List<Integer>>
                    partitionPrimesWithCustomCollector(int n) {
     return IntStream.rangeClosed(2, n).boxed()
                    .collect(new PrimeNumbersCollector());
}
```

6.6.2 比较收集器的性能

用partitioningBy工厂方法创建的收集器和你刚刚开发的自定义收集器在功能上是一样的，但是我们有没有实现用自定义收集器超越partitioningBy收集器性能的目标呢？现在让我们写个小测试框架来跑一下吧：

```
public class CollectorHarness {
    public static void main(String[] args) {
        long fastest = Long.MAX_VALUE;
        for (int i = 0; i < 10; i++) {
            long start = System.nanoTime();
            partitionPrimes(1_000_000);
            long duration = (System.nanoTime() - start) / 1_000_000;
            if (duration < fastest) fastest = duration;
        }
        System.out.println(
            "Fastest execution done in " + fastest + " msecs");
    }
}
```

运行测试10次

将前一百万个自然数按质数和非质数分区

取运行时间的毫秒值

检查这个执行是否是最快的一个

请注意，更为科学的测试方法是用一个诸如JMH的框架，但我们不想在这里把问题搞得更复杂。对这个例子而言，这个小小的测试类提供的结果足够准确了。这个类会先把前一百万个自然

数分为质数和非质数，利用partitioningBy工厂方法创建的收集器调用方法10次，记下最快的一次运行。在英特尔i5 2.4 GHz的机器上运行得到了以下结果：

```
Fastest execution done in 4716 msecs
```

现在把测试框架的partitionPrimes换成partitionPrimesWithCustomCollector，以便测试我们开发的自定义收集器的性能。现在，程序打印：

```
Fastest execution done in 3201 msecs
```

还不错！这意味着开发自定义收集器并不是白费工夫，原因有二：第一，你学会了如何在需要的时候实现自己的收集器；第二，你获得了大约32%的性能提升。

最后还有一点很重要，就像代码清单6-5中的ToListCollector那样，也可以通过把实现PrimeNumbersCollector核心逻辑的三个函数传给collect方法的重载版本来获得同样的结果：

```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
        (int n) {
    IntStream.rangeClosed(2, n).boxed()
        .collect(
            () -> new HashMap<Boolean, List<Integer>>() {{          // 供应源
                put(true, new ArrayList<Integer>());
                put(false, new ArrayList<Integer>());
            }},
            (acc, candidate) -> {                                   // 累加器
                acc.get( isPrime(acc.get(true), candidate) )
                   .add(candidate);
            },
            (map1, map2) -> {                                       // 组合器
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
            });
}
```

你看，这样就可以避免为实现Collector接口创建一个全新的类；得到的代码更紧凑，虽然可能可读性会差一点，可重用性会差一点。

6.7　小结

以下是你应从本章中学到的关键概念。

- collect是一个终端操作，它接受的参数是将流中元素累积到汇总结果的各种方式（称为收集器）。
- 预定义收集器包括将流元素归约和汇总到一个值，例如计算最小值、最大值或平均值。这些收集器总结在表6-1中。
- 预定义收集器可以用groupingBy对流中元素进行分组，或用partitioningBy进行分区。
- 收集器可以高效地复合起来，进行多级分组、分区和归约。
- 你可以实现Collector接口中定义的方法来开发你自己的收集器。

第 7 章 并行数据处理与性能

本章内容

- ❑ 用并行流并行处理数据
- ❑ 并行流的性能分析
- ❑ 分支/合并框架
- ❑ 使用Spliterator分割流

在前面三章中，我们已经看到了新的Stream接口可以让你以声明性方式处理数据集。我们还解释了将外部迭代换为内部迭代能够让原生Java库控制流元素的处理。这种方法让Java程序员无需显式实现优化来为数据集的处理加速。到目前为止，最重要的好处是可以对这些集合执行操作流水线，能够自动利用计算机上的多个内核。

例如，在Java 7之前，并行处理数据集合非常麻烦。第一，你得明确地把包含数据的数据结构分成若干子部分。第二，你要给每个子部分分配一个独立的线程。第三，你需要在恰当的时候对它们进行同步来避免不希望出现的竞争条件，等待所有线程完成，最后把这些部分结果合并起来。Java 7引入了一个叫作分支/合并的框架，让这些操作更稳定、更不易出错。我们会在7.2节探讨这一框架。

在本章中，你将了解Stream接口如何让你不用太费力气就能对数据集执行并行操作。它允许你声明性地将顺序流变为并行流。此外，你将看到Java是如何变戏法的，或者更实际地来说，流是如何在幕后应用Java 7引入的分支/合并框架的。你还会发现，了解并行流内部是如何工作的很重要，因为如果你忽视这一方面，就可能因误用而得到意外的（很可能是错的）结果。

我们会特别演示，在并行处理数据块之前，并行流被划分为数据块的方式在某些情况下恰恰是这些错误且无法解释的结果的根源。因此，你将会学习如何通过实现和使用你自己的Spliterator来控制这个划分过程。

7.1 并行流

在第4章中，我们简要地提到了Stream接口可以让你非常方便地处理它的元素：可以通过对收集源调用parallelStream方法来把集合转换为并行流。并行流就是一个把内容分成多个数据

块，并用不同的线程分别处理每个数据块的流。这样一来，你就可以自动把给定操作的工作负荷分配给多核处理器的所有内核，让它们都忙起来。让我们用一个简单的例子来试验一下这个思想。

假设你需要写一个方法，接受数字*n*作为参数，并返回从1到给定参数的所有数字的和。一个直接（也许有点土）的方法是生成一个无穷大的数字流，把它限制到给定的数目，然后用对两个数字求和的BinaryOperator来归约这个流，如下所示：

```
public static long sequentialSum(long n) {
    return Stream.iterate(1L, i -> i + 1)        // 生成自然数无限流
                 .limit(n)                        // 限制到前n个数
                 .reduce(0L, Long::sum);          // 对所有数字求和来归纳流
}
```

用更为传统的Java术语来说，这段代码与下面的迭代等价：

```
public static long iterativeSum(long n) {
    long result = 0;
    for (long i = 1L; i <= n; i++) {
        result += i;
    }
    return result;
}
```

这似乎是利用并行处理的好机会，特别是*n*很大的时候。那怎么入手呢？你要对结果变量进行同步吗？用多少个线程呢？谁负责生成数呢？谁来做加法呢？

根本用不着担心啦。用并行流的话，这问题就简单多了！

7.1.1　将顺序流转换为并行流

你可以把流转换成并行流，从而让前面的函数归约过程（也就是求和）并行运行——对顺序流调用parallel方法：

```
public static long parallelSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
                 .limit(n)
                 .parallel()                      // 将流转换为并行流
                 .reduce(0L, Long::sum);
}
```

在上面的代码中，对流中所有数字求和的归纳过程的执行方式和5.4.1节中说的差不多。不同之处在于Stream在内部分成了几块。因此可以对不同的块独立并行进行归纳操作，如图7-1所示。最后，同一个归纳操作会将各个子流的部分归纳结果合并起来，得到整个原始流的归纳结果。

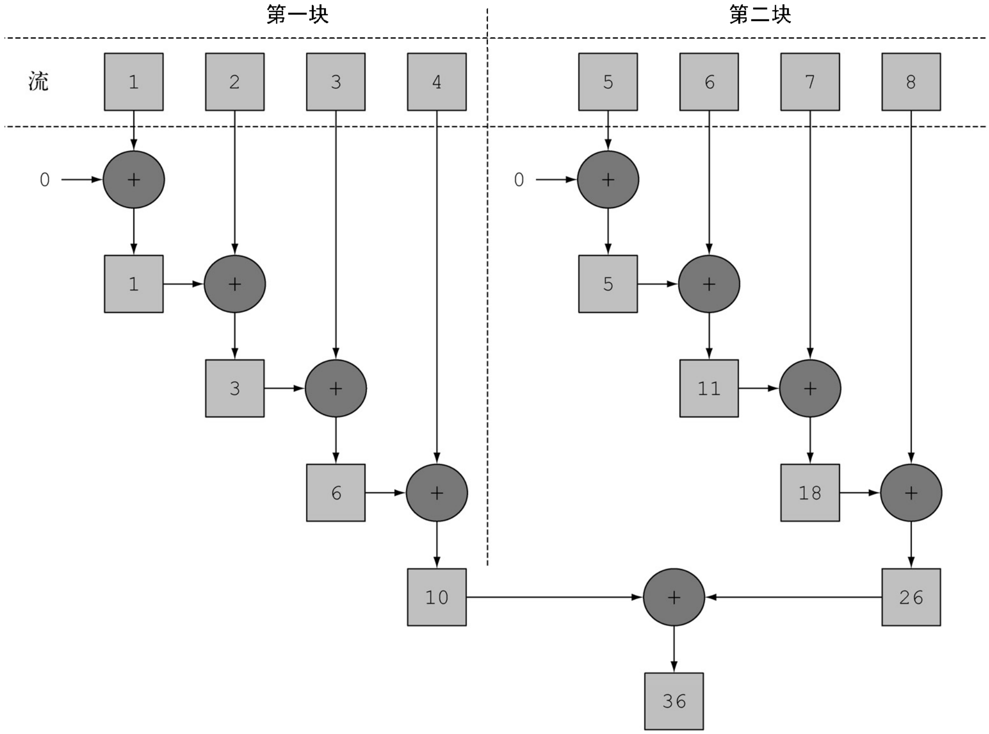


图7-1　并行归纳操作

请注意，在现实中，对顺序流调用parallel方法并不意味着流本身有任何实际的变化。它在内部实际上就是设了一个boolean标志，表示你想让调用parallel之后进行的所有操作都并行执行。类似地，你只需要对并行流调用sequential方法就可以把它变成顺序流。请注意，你可能以为把这两个方法结合起来，就可以更细化地控制在遍历流时哪些操作要并行执行，哪些要顺序执行。例如，你可以这样做：

```
stream.parallel()
      .filter(...)
      .sequential()
      .map(...)
      .parallel()
      .reduce();
```

但最后一次parallel或sequential调用会影响整个流水线。在本例中，流水线会并行执行，因为最后调用的是它。

配置并行流使用的线程池

看看流的parallel方法，你可能会想，并行流用的线程是从哪儿来的？有多少个？怎么自定义这个过程呢？

> 并行流内部使用了默认的ForkJoinPool（7.2节会进一步讲到分支/合并框架），它默认的线程数量就是你的处理器数量，这个值是由Runtime.getRuntime().availableProcessors()得到的。
>
> 但是你可以通过系统属性java.util.concurrent.ForkJoinPool.common.parallelism来改变线程池大小，如下所示：
>
> ```
> System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism","12");
> ```
>
> 这是一个全局设置，因此它将影响代码中所有的并行流。反过来说，目前还无法专为某个并行流指定这个值。一般而言，让ForkJoinPool的大小等于处理器数量是个不错的默认值，除非你有很好的理由，否则我们强烈建议你不要修改它。

回到我们的数字求和练习，我们说过，在多核处理器上运行并行版本时，会有显著的性能提升。现在你有三个方法，用三种不同的方式（迭代式、顺序归纳和并行归纳）做完全相同的操作，让我们看看谁最快吧！

7.1.2　测量流性能

我们声称并行求和方法应该比顺序和迭代方法性能好。然而在软件工程上，靠猜绝对不是什么好办法！特别是在优化性能时，你应该始终遵循三个黄金规则：测量，测量，再测量。为此，你可以开发一个方法，它与6.6.2节中用于比较划分质数的两个收集器性能的测试框架非常类似，如下所示。

代码清单7-1　测量对前*n*个自然数求和的函数的性能

```
public long measureSumPerf(Function<Long, Long> adder, long n) {
    long fastest = Long.MAX_VALUE;
    for (int i = 0; i < 10; i++) {
        long start = System.nanoTime();
        long sum = adder.apply(n);
        long duration = (System.nanoTime() - start) / 1_000_000;
        System.out.println("Result: " + sum);
        if (duration < fastest) fastest = duration;
    }
    return fastest;
}
```

这个方法接受一个函数和一个long作为参数。它会对传给方法的long应用函数10次，记录每次执行的时间（以毫秒为单位），并返回最短的一次执行时间。假设你把先前开发的所有方法都放进了一个名为ParallelStreams的类，你就可以用这个框架来测试顺序加法器函数对前一千万个自然数求和要用多久：

```
System.out.println("Sequential sum done in:" +
    measureSumPerf(ParallelStreams::sequentialSum, 10_000_000) + " msecs");
```

请注意，我们对这个结果应持保留态度。影响执行时间的因素有很多，比如你的电脑支持多少个内核。你可以在自己的机器上跑一下这些代码。我们在一台四核英特尔i7 2.3 GHz的MacBook Pro上运行它，输出是这样的：

```
Sequential sum done in: 97 msecs
```

用传统for循环的迭代版本执行起来应该会快很多，因为它更为底层，更重要的是不需要对原始类型做任何装箱或拆箱操作。如果你试着测量它的性能，

```
System.out.println("Iterative sum done in:" +
    measureSumPerf(ParallelStreams::iterativeSum, 10_000_000) + " msecs");
```

将得到：

```
Iterative sum done in: 2 msecs
```

现在我们来对函数的并行版本做测试：

```
System.out.println("Parallel sum done in: " +
    measureSumPerf(ParallelStreams::parallelSum, 10_000_000) + " msecs" );
```

看看会出现什么情况：

```
Parallel sum done in: 164 msecs
```

这相当令人失望，求和方法的并行版本比顺序版本要慢很多。你如何解释这个意外的结果呢？这里实际上有两个问题：

- iterate生成的是装箱的对象，必须拆箱成数字才能求和；
- 我们很难把iterate分成多个独立块来并行执行。

第二个问题更有意思一点，因为你必须意识到某些流操作比其他操作更容易并行化。具体来说，iterate很难分割成能够独立执行的小块，因为每次应用这个函数都要依赖前一次应用的结果，如图7-2所示。

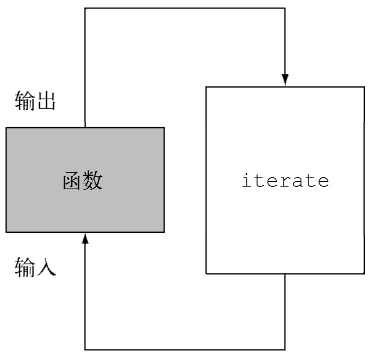


图7-2　iterate在本质上是顺序的

这意味着，在这个特定情况下，归纳进程不是像图7-1那样进行的；整张数字列表在归纳过

程开始时没有准备好，因而无法有效地把流划分为小块来并行处理。把流标记成并行，你其实是给顺序处理增加了开销，它还要把每次求和操作分到一个不同的线程上。

这就说明了并行编程可能很复杂，有时候甚至有点违反直觉。如果用得不对（比如采用了一个不易并行化的操作，如`iterate`），它甚至可能让程序的整体性能更差，所以在调用那个看似神奇的`parallel`操作时，了解背后到底发生了什么是很有必要的。

使用更有针对性的方法

那到底要怎么利用多核处理器，用流来高效地并行求和呢？我们在第5章中讨论了一个叫`LongStream.rangeClosed`的方法。这个方法与`iterate`相比有两个优点。

- `LongStream.rangeClosed`直接产生原始类型的`long`数字，没有装箱拆箱的开销。
- `LongStream.rangeClosed`会生成数字范围，很容易拆分为独立的小块。例如，范围1~20可分为1~5、6~10、11~15和16~20。

让我们先看一下它用于顺序流时的性能如何，看看拆箱的开销到底要不要紧：

```
public static long rangedSum(long n) {
    return LongStream.rangeClosed(1, n)
                     .reduce(0L, Long::sum);
}
```

这一次的输出是：

```
Ranged sum done in: 17 msecs
```

这个数值流比前面那个用`iterate`工厂方法生成数字的顺序执行版本要快得多，因为数值流避免了非针对性流那些没必要的自动装箱和拆箱操作。由此可见，选择适当的数据结构往往比并行化算法更重要。但要是对这个新版本应用并行流呢？

```
public static long parallelRangedSum(long n) {
    return LongStream.rangeClosed(1, n)
                     .parallel()
                     .reduce(0L, Long::sum);
}
```

现在把这个函数传给你的测试方法：

```
System.out.println("Parallel range sum done in:" +
    measureSumPerf(ParallelStreams::parallelRangedSum, 10_000_000) +
    " msecs");
```

你会得到：

```
Parallel range sum done in: 1 msecs
```

终于，我们得到了一个比顺序执行更快的并行归纳，因为这一次归纳操作可以像图7-1那样执行了。这也表明，使用正确的数据结构然后使其并行工作能够保证最佳的性能。

尽管如此，请记住，并行化并不是没有代价的。并行化过程本身需要对流做递归划分，把每个子流的归纳操作分配到不同的线程，然后把这些操作的结果合并成一个值。但在多个内核之间移动数据的代价也可能比你想的要大，所以很重要的一点是要保证在内核中并行执行工作的时间

比在内核之间传输数据的时间长。总而言之，很多情况下不可能或不方便并行化。然而，在使用并行Stream加速代码之前，你必须确保用得对；如果结果错了，算得快就毫无意义了。让我们来看一个常见的陷阱。

7.1.3 正确使用并行流

错用并行流而产生错误的首要原因，就是使用的算法改变了某些共享状态。下面是另一种实现对前*n*个自然数求和的方法，但这会改变一个共享累加器：

```
public static long sideEffectSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).forEach(accumulator::add);
    return accumulator.total;
}
public class Accumulator {
    public long total = 0;
    public void add(long value) { total += value; }
}
```

这种代码非常普遍，特别是对那些熟悉指令式编程范式的程序员来说。这段代码和你习惯的那种指令式迭代数字列表的方式很像：初始化一个累加器，一个个遍历列表中的元素，把它们和累加器相加。

那这种代码又有什么问题呢？不幸的是，它真的无可救药，因为它在本质上就是顺序的。每次访问total都会出现数据竞争。如果你尝试用同步来修复，那就完全失去并行的意义了。为了说明这一点，让我们试着把Stream变成并行的：

```
public static long sideEffectParallelSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);
    return accumulator.total;
}
```

用代码清单7-1中的测试框架来执行这个方法，并打印每次执行的结果：

```
System.out.println("SideEffect parallel sum done in: " +
    measurePerf(ParallelStreams::sideEffectParallelSum, 10_000_000L) +"
    msecs" );
```

你可能会得到类似于下面这种输出：

```
Result: 5959989000692
Result: 7425264100768
Result: 6827235020033
Result: 7192970417739
Result: 6714157975331
Result: 7497810541907
Result: 6435348440385
Result: 6999349840672
Result: 7435914379978
```

```
Result: 7715125932481
SideEffect parallel sum done in: 49 msecs
```

这回方法的性能无关紧要了，唯一要紧的是每次执行都会返回不同的结果，都离正确值50000005000000差很远。这是由于多个线程在同时访问累加器，执行total += value，而这一句虽然看似简单，却不是一个原子操作。问题的根源在于，forEach中调用的方法有副作用，它会改变多个线程共享的对象的可变状态。要是你想用并行Stream又不想引发类似的意外，就必须避免这种情况。

现在你知道了，共享可变状态会影响并行流以及并行计算。第13章和第14章详细讨论函数式编程的时候，我们还会谈到这一点。现在，记住要避免共享可变状态，确保并行Stream得到正确的结果。接下来，我们会看到一些实用建议，你可以由此判断什么时候可以利用并行流来提升性能。

7.1.4　高效使用并行流

一般而言，想给出任何关于什么时候该用并行流的定量建议都是不可能也毫无意义的，因为任何类似于“仅当至少有一千个（或一百万个或随便什么数字）元素的时候才用并行流）”的建议对于某台特定机器上的某个特定操作可能是对的，但在略有差异的另一种情况下可能就是大错特错。尽管如此，我们至少可以提出一些定性意见，帮你决定某个特定情况下是否有必要使用并行流。

- 如果有疑问，测量。把顺序流转成并行流轻而易举，但却不一定是好事。我们在本节中已经指出，并行流并不总是比顺序流快。此外，并行流有时候会和你的直觉不一致，所以在考虑选择顺序流还是并行流时，第一个也是最重要的建议就是用适当的基准来检查其性能。
- 留意装箱。自动装箱和拆箱操作会大大降低性能。Java 8中有原始类型流（IntStream、LongStream、DoubleStream）来避免这种操作，但凡有可能都应该用这些流。
- 有些操作本身在并行流上的性能就比顺序流差。特别是limit和findFirst等依赖于元素顺序的操作，它们在并行流上执行的代价非常大。例如，findAny会比findFirst性能好，因为它不一定要按顺序来执行。你总是可以调用unordered方法来把有序流变成无序流。那么，如果你需要流中的*n*个元素而不是专门要前*n*个的话，对无序并行流调用limit可能会比单个有序流（比如数据源是一个List）更高效。
- 还要考虑流的操作流水线的总计算成本。设*N*是要处理的元素的总数，*Q*是一个元素通过流水线的大致处理成本，则*N***Q*就是这个对成本的一个粗略的定性估计。*Q*值较高就意味着使用并行流时性能好的可能性比较大。
- 对于较小的数据量，选择并行流几乎从来都不是一个好的决定。并行处理少数几个元素的好处还抵不上并行化造成的额外开销。
- 要考虑流背后的数据结构是否易于分解。例如，ArrayList的拆分效率比LinkedList高得多，因为前者用不着遍历就可以平均拆分，而后者则必须遍历。另外，用range工厂

方法创建的原始类型流也可以快速分解。最后，你将在7.3节中学到，你可以自己实现Spliterator来完全掌控分解过程。
- 流自身的特点，以及流水线中的中间操作修改流的方式，都可能会改变分解过程的性能。例如，一个SIZED流可以分成大小相等的两部分，这样每个部分都可以比较高效地并行处理，但筛选操作可能丢弃的元素个数却无法预测，导致流本身的大小未知。
- 还要考虑终端操作中合并步骤的代价是大是小（例如Collector中的combiner方法）。如果这一步代价很大，那么组合每个子流产生的部分结果所付出的代价就可能会超出通过并行流得到的性能提升。

表7-1按照可分解性总结了一些流数据源适不适于并行。

表7-1　流的数据源和可分解性

源	可分解性
ArrayList	极佳
LinkedList	差
IntStream.range	极佳
Stream.iterate	差
HashSet	好
TreeSet	好

最后，我们还要强调并行流背后使用的基础架构是Java 7中引入的分支/合并框架。并行汇总的示例证明了要想正确使用并行流，了解它的内部原理至关重要，所以我们会在下一节仔细研究分支/合并框架。

7.2　分支/合并框架

分支/合并框架的目的是以递归方式将可以并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果。它是ExecutorService接口的一个实现，它把子任务分配给线程池（称为ForkJoinPool）中的工作线程。首先来看看如何定义任务和子任务。

7.2.1　使用 RecursiveTask

要把任务提交到这个池，必须创建RecursiveTask<R>的一个子类，其中R是并行化任务（以及所有子任务）产生的结果类型，或者如果任务不返回结果，则是RecursiveAction类型（当然它可能会更新其他非局部机构）。要定义RecursiveTask，只需实现它唯一的抽象方法compute：

```
protected abstract R compute();
```

这个方法同时定义了将任务拆分成子任务的逻辑，以及无法再拆分或不方便再拆分时，生成单个子任务结果的逻辑。正由于此，这个方法的实现类似于下面的伪代码：

```
if (任务足够小或不可分) {
    顺序计算该任务
} else {
    将任务分成两个子任务
    递归调用本方法，拆分每个子任务，等待所有子任务完成
    合并每个子任务的结果
}
```

一般来说并没有确切的标准决定一个任务是否应该再拆分，但有几种试探方法可以帮助你做出这一决定。我们会在7.2.1节中进一步澄清。递归的任务拆分过程如图7-3所示。

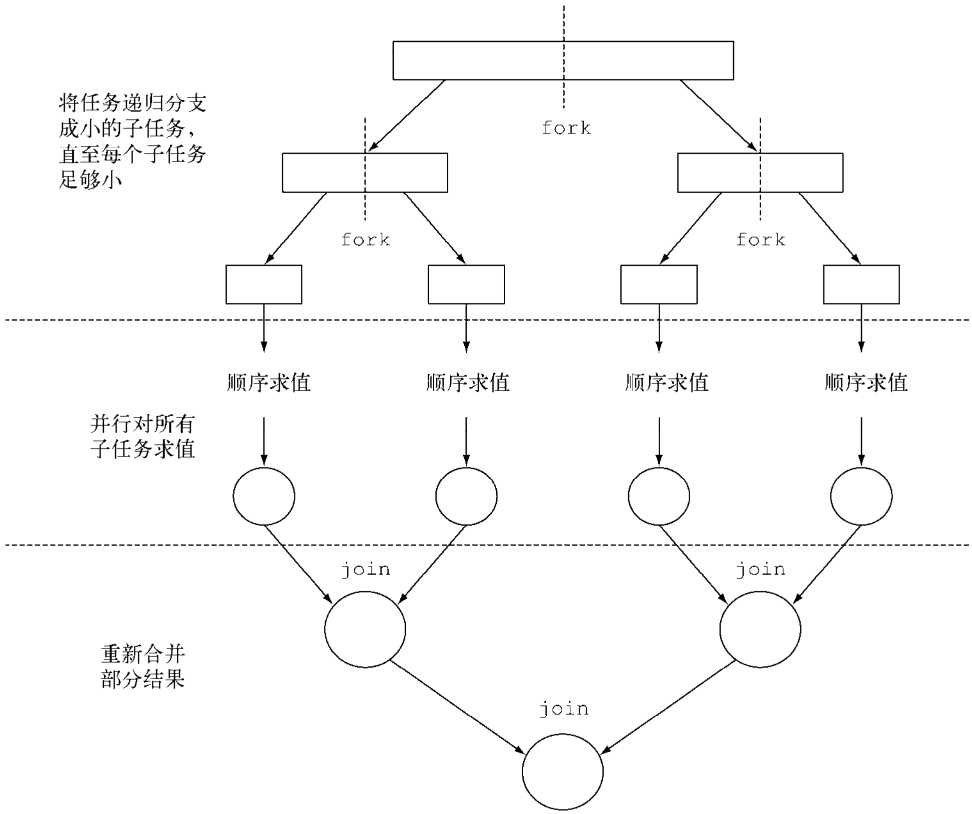


图7-3　分支/合并过程

你可能已经注意到，这只不过是著名的分治算法的并行版本而已。这里举一个用分支/合并框架的实际例子，还以前面的例子为基础，让我们试着用这个框架为一个数字范围（这里用一个`long[]`数组表示）求和。如前所述，你需要先为`RecursiveTask`类做一个实现，就是下面代码清单中的`ForkJoinSumCalculator`。

代码清单7-2 用分支/合并框架执行并行求和

```
public class ForkJoinSumCalculator
        extends java.util.concurrent.RecursiveTask<Long> {    ← 继承RecursiveTask来创建可以用于分支/合并框架的任务

    private final long[] numbers;    ← 要求和的数组
    private final int start;         ← 子任务处理的数组的起始和终止位置
    private final int end;

    public static final long THRESHOLD = 10_000;    ← 不再将任务分解为子任务的数组大小

    public ForkJoinSumCalculator(long[] numbers) {    ← 公共构造函数用于创建主任务
        this(numbers, 0, numbers.length);
    }

    private ForkJoinSumCalculator(long[] numbers, int start, int end) {    ← 私有构造函数用于以递归方式为主任务创建子任务
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {    ← 覆盖RecursiveTask抽象方法
        int length = end - start;    ← 该任务负责求和的部分的大小
        if (length <= THRESHOLD) {
            return computeSequentially();    ← 如果大小小于或等于阈值，顺序计算结果
        }
        ForkJoinSumCalculator leftTask =
            new ForkJoinSumCalculator(numbers, start, start + length/2);    ← 创建一个子任务来为数组的前一半求和
        leftTask.fork();    ← 利用另一个ForkJoinPool线程异步执行新创建的子任务
        ForkJoinSumCalculator rightTask =
            new ForkJoinSumCalculator(numbers, start + length/2, end);    ← 创建一个任务为数组的后一半求和
        Long rightResult = rightTask.compute();    ← 同步执行第二个子任务，有可能允许进一步递归划分
        Long leftResult = leftTask.join();    ← 读取第一个子任务的结果，如果尚未完成就等待
        return leftResult + rightResult;    ← 该任务的结果是两个子任务结果的组合
    }

    private long computeSequentially() {    ← 在子任务不再可分时计算结果的简单算法
        long sum = 0;
        for (int i = start; i < end; i++) {
            sum += numbers[i];
        }
        return sum;
    }
}
```

现在编写一个方法来并行对前*n*个自然数求和就很简单了。你只需把想要的数字数组传给`ForkJoinSumCalculator`的构造函数：

```
public static long forkJoinSum(long n) {
    long[] numbers = LongStream.rangeClosed(1, n).toArray();
    ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);
    return new ForkJoinPool().invoke(task);
}
```

这里用了一个LongStream来生成包含前*n*个自然数的数组，然后创建一个ForkJoinTask（RecursiveTask的父类），并把数组传递给代码清单7-2所示ForkJoinSumCalculator的公共构造函数。最后，你创建了一个新的ForkJoinPool，并把任务传给它的调用方法 。在ForkJoinPool中执行时，最后一个方法返回的值就是ForkJoinSumCalculator类定义的任务结果。

请注意在实际应用时，使用多个ForkJoinPool是没有什么意义的。正是出于这个原因，一般来说把它实例化一次，然后把实例保存在静态字段中，使之成为单例，这样就可以在软件中任何部分方便地重用了。这里创建时用了其默认的无参数构造函数，这意味着想让线程池使用JVM能够使用的所有处理器。更确切地说，该构造函数将使用Runtime.availableProcessors的返回值来决定线程池使用的线程数。请注意availableProcessors方法虽然看起来是处理器，但它实际上返回的是可用内核的数量，包括超线程生成的虚拟内核。

运行ForkJoinSumCalculator

当把ForkJoinSumCalculator任务传给ForkJoinPool时，这个任务就由池中的一个线程执行，这个线程会调用任务的compute方法。该方法会检查任务是否小到足以顺序执行，如果不够小则会把要求和的数组分成两半，分给两个新的ForkJoinSumCalculator，而它们也由ForkJoinPool安排执行。因此，这一过程可以递归重复，把原任务分为更小的任务，直到满足不方便或不可能再进一步拆分的条件（本例中是求和的项目数小于等于10 000）。这时会顺序计算每个任务的结果，然后由分支过程创建的（隐含的）任务二叉树遍历回到它的根。接下来会合并每个子任务的部分结果，从而得到总任务的结果。这一过程如图7-4所示。

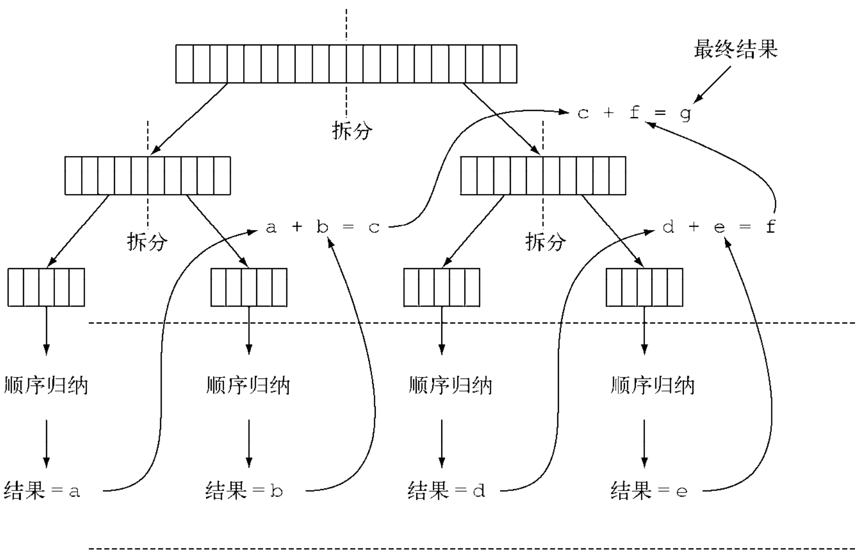


图7-4　分支/合并算法

你可以再用一次本章开始时写的测试框架，来看看显式使用分支/合并框架的求和方法的性能：

```
System.out.println("ForkJoin sum done in: " + measureSumPerf(
        ForkJoinSumCalculator::forkJoinSum, 10_000_000) + " msecs" );
```

它生成以下输出：

```
ForkJoin sum done in: 41 msecs
```

这个性能看起来比用并行流的版本要差，但这只是因为必须先要把整个数字流都放进一个long[]，之后才能在ForkJoinSumCalculator任务中使用它。

7.2.2 使用分支/合并框架的最佳做法

虽然分支/合并框架还算简单易用，不幸的是它也很容易被误用。以下是几个有效使用它的最佳做法。

- 对一个任务调用join方法会阻塞调用方，直到该任务做出结果。因此，有必要在两个子任务的计算都开始之后再调用它。否则，你得到的版本会比原始的顺序算法更慢更复杂，因为每个子任务都必须等待另一个子任务完成才能启动。
- 不应该在RecursiveTask内部使用ForkJoinPool的invoke方法。相反，你应该始终直接调用compute或fork方法，只有顺序代码才应该用invoke来启动并行计算。
- 对子任务调用fork方法可以把它排进ForkJoinPool。同时对左边和右边的子任务调用它似乎很自然，但这样做的效率要比直接对其中一个调用compute低。这样做你可以为其中一个子任务重用同一线程，从而避免在线程池中多分配一个任务造成的开销。
- 调试使用分支/合并框架的并行计算可能有点棘手。特别是你平常都在你喜欢的IDE里面看栈跟踪（stack trace）来找问题，但放在分支–合并计算上就不行了，因为调用compute的线程并不是概念上的调用方，后者是调用fork的那个。
- 和并行流一样，你不应理所当然地认为在多核处理器上使用分支/合并框架就比顺序计算快。我们已经说过，一个任务可以分解成多个独立的子任务，才能让性能在并行化时有所提升。所有这些子任务的运行时间都应该比分出新任务所花的时间长；一个惯用方法是把输入/输出放在一个子任务里，计算放在另一个里，这样计算就可以和输入/输出同时进行。此外，在比较同一算法的顺序和并行版本的性能时还有别的因素要考虑。就像任何其他Java代码一样，分支/合并框架需要“预热”或者说要执行几遍才会被JIT编译器优化。这就是为什么在测量性能之前跑几遍程序很重要，我们的测试框架就是这么做的。同时还要知道，编译器内置的优化可能会为顺序版本带来一些优势（例如执行死码分析——删去从未被使用的计算）。

对于分支/合并拆分策略还有最后一点补充：你必须选择一个标准，来决定任务是要进一步拆分还是已小到可以顺序求值。我们会在下一节中就此给出一些提示。

7.2.3 工作窃取

在ForkJoinSumCalculator的例子中，我们决定在要求和的数组中最多包含10 000个项目时就不再创建子任务了。这个选择是很随意的，但大多数情况下也很难找到一个好的启发式方法来确定它，只能试几个不同的值来尝试优化它。在我们的测试案例中，我们先用了一个有1000万项目的数组，意味着ForkJoinSumCalculator至少会分出1000个子任务来。这似乎有点浪费资源，因为我们用来运行它的机器上只有四个内核。在这个特定例子中可能确实是这样，因为所有的任务都受CPU约束，预计所花的时间也差不多。

但分出大量的小任务一般来说都是一个好的选择。这是因为，理想情况下，划分并行任务时，应该让每个任务都用完全相同的时间完成，让所有的CPU内核都同样繁忙。不幸的是，实际中，每个子任务所花的时间可能天差地别，要么是因为划分策略效率低，要么是有不可预知的原因，比如磁盘访问慢，或是需要和外部服务协调执行。

分支/合并框架工程用一种称为*工作窃取*（work stealing）的技术来解决这个问题。在实际应用中，这意味着这些任务差不多被平均分配到ForkJoinPool中的所有线程上。每个线程都为分配给它的任务保存一个双向链式队列，每完成一个任务，就会从队列头上取出下一个任务开始执行。基于前面所述的原因，某个线程可能早早完成了分配给它的所有任务，也就是它的队列已经空了，而其他的线程还很忙。这时，这个线程并没有闲下来，而是随机选了一个别的线程，从队列的尾巴上"偷走"一个任务。这个过程一直继续下去，直到所有的任务都执行完毕，所有的队列都清空。这就是为什么要划成许多小任务而不是少数几个大任务，这有助于更好地在工作线程之间平衡负载。

一般来说，这种工作窃取算法用于在池中的工作线程之间重新分配和平衡任务。图7-5展示了这个过程。当工作线程队列中有一个任务被分成两个子任务时，一个子任务就被闲置的工作线程"偷走"了。如前所述，这个过程可以不断递归，直到规定子任务应顺序执行的条件为真。

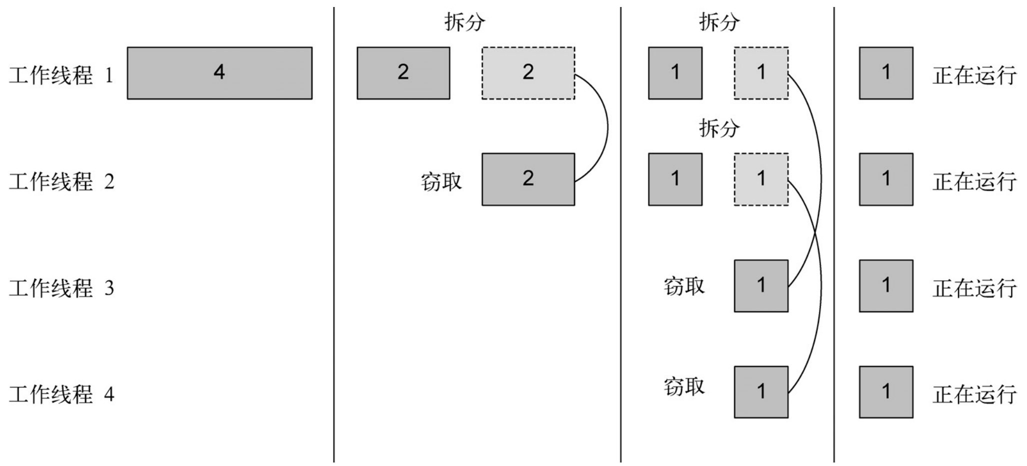


图7-5 分支/合并框架使用的工作窃取算法

现在你应该清楚流如何使用分支/合并框架来并行处理它的项目了，不过还有一点没有讲。本节中我们分析了一个例子，你明确地指定了将数字数组拆分成多个任务的逻辑。但是，使用本章前面讲的并行流时就用不着这么做了，这就意味着，肯定有一种自动机制来为你拆分流。这种新的自动机制称为Spliterator，我们会在下一节中讨论。

7.3 **Spliterator**

Spliterator是Java 8中加入的另一个新接口；这个名字代表“可分迭代器”（splitable iterator）。和Iterator一样，Spliterator也用于遍历数据源中的元素，但它是为了并行执行而设计的。虽然在实践中可能用不着自己开发Spliterator，但了解一下它的实现方式会让你对并行流的工作原理有更深入的了解。Java 8已经为集合框架中包含的所有数据结构提供了一个默认的Spliterator实现。集合实现了Spliterator接口，接口提供了一个spliterator方法。这个接口定义了若干方法，如下面的代码清单所示。

代码清单7-3 Spliterator接口

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

与往常一样，T是Spliterator遍历的元素的类型。tryAdvance方法的行为类似于普通的Iterator，因为它会按顺序一个一个使用Spliterator中的元素，并且如果还有其他元素要遍历就返回true。但trySplit是专为Spliterator接口设计的，因为它可以把一些元素划出去分给第二个Spliterator（由该方法返回），让它们两个并行处理。Spliterator还可通过estimateSize方法估计还剩下多少元素要遍历，因为即使不那么确切，能快速算出来是一个值也有助于让拆分均匀一点。

重要的是，要了解这个拆分过程在内部是如何执行的，以便在需要时能够掌控它。因此，我们会在下一节中详细地分析它。

7.3.1 拆分过程

将Stream拆分成多个部分的算法是一个递归过程，如图7-6所示。第一步是对第一个Spliterator调用trySplit，生成第二个Spliterator。第二步对这两个Spliterator调用trysplit，这样总共就有了四个Spliterator。这个框架不断对Spliterator调用trySplit直到它返回null，表明它处理的数据结构不能再分割，如第三步所示。最后，这个递归拆分过程到第四步就终止了，这时所有的Spliterator在调用trySplit时都返回了null。

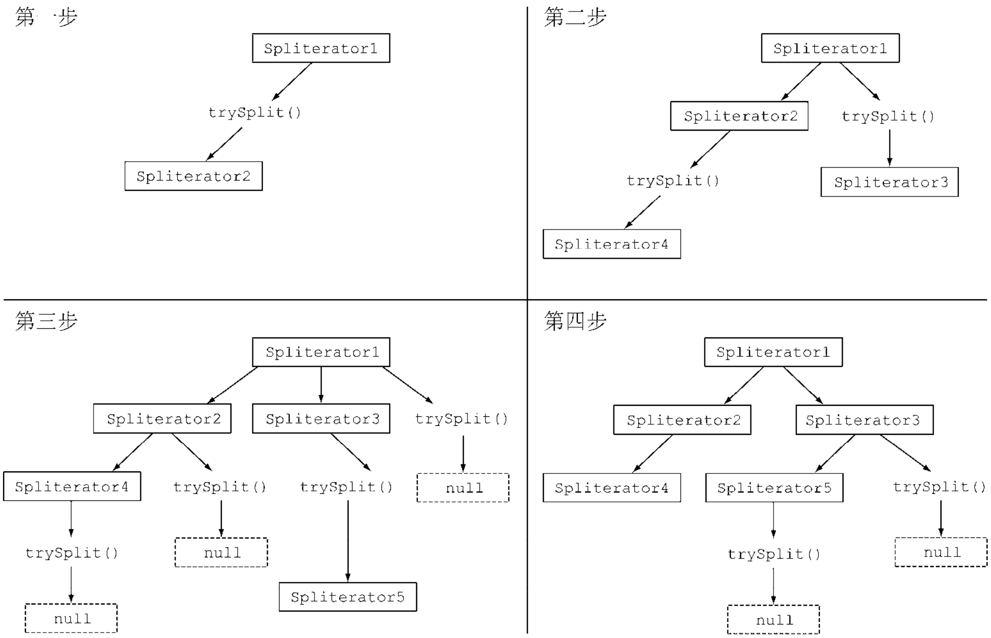


图7-6　递归拆分过程

这个拆分过程也受Spliterator本身的特性影响，而特性是通过characteristics方法声明的。

Spliterator的特性

Spliterator接口声明的最后一个抽象方法是characteristics，它将返回一个int，代表Spliterator本身特性集的编码。使用Spliterator的客户可以用这些特性来更好地控制和优化它的使用。表7-2总结了这些特性。（不幸的是，虽然它们在概念上与收集器的特性有重叠，编码却不一样。）

表7-2　Spliterator的特性

特　　性	含　　义
ORDERED	元素有既定的顺序（例如List），因此Spliterator在遍历和划分时也会遵循这一顺序
DISTINCT	对于任意一对遍历过的元素x和y，x.equals(y)返回false
SORTED	遍历的元素按照一个预定义的顺序排序
SIZED	该Spliterator由一个已知大小的源建立（例如Set），因此estimatedSize()返回的是准确值
NONNULL	保证遍历的元素不会为null
IMMUTABLE	Spliterator的数据源不能修改。这意味着在遍历时不能添加、删除或修改任何元素
CONCURRENT	该Spliterator的数据源可以被其他线程同时修改而无需同步
SUBSIZED	该Spliterator和所有从它拆分出来的Spliterator都是SIZED

7.3 Spliterator

现在你已经看到了Spliterator接口是什么以及它定义了哪些方法，你可以试着自己实现一个Spliterator了。

7.3.2 实现你自己的 Spliterator

让我们来看一个可能需要你自己实现Spliterator的实际例子。我们要开发一个简单的方法来数数一个String中的单词数。这个方法的一个迭代版本可以写成下面的样子。

代码清单7-4 一个迭代式字数统计方法

```
public int countWordsIteratively(String s) {
    int counter = 0;
    boolean lastSpace = true;
    for (char c : s.toCharArray()) {            <-- 逐个遍历String中的所有字符
        if (Character.isWhitespace(c)) {
            lastSpace = true;
        } else {
            if (lastSpace) counter++;          <-- 上一个字符是空格，而当前遍历的字符不是空格时，将单词计数器加一
            lastSpace = false;
        }
    }
    return counter;
}
```


让我们把这个方法用在但丁的《神曲》的《地狱篇》的第一句话上：[①]

```
final String SENTENCE =
            " Nel   mezzo del cammin  di nostra  vita " +
            "mi  ritrovai in una  selva oscura" +
            " ché la  dritta via era   smarrita ";

System.out.println("Found " + countWordsIteratively(SENTENCE) + " words");
```

请注意，我们在句子里添加了一些额外的随机空格，以演示这个迭代实现即使在两个词之间存在多个空格时也能正常工作。正如我们所料，这段代码将打印以下内容：

```
Found 19 words
```

理想情况下，你会想要用更为函数式的风格来实现它，因为就像我们前面说过的，这样你就可以用并行Stream来并行化这个过程，而无需显式地处理线程和同步问题。

1. 以函数式风格重写单词计数器

首先你需要把String转换成一个流。不幸的是，原始类型的流仅限于int、long和double，所以你只能用Stream<Character>：

```
Stream<Character> stream = IntStream.range(0, SENTENCE.length())
                                    .mapToObj(SENTENCE::charAt);
```

你可以对这个流做归约来计算字数。在归约流时，你得保留由两个变量组成的状态：一个int

① 请参阅http://en.wikipedia.org/wiki/Inferno_(Dante)。

用来计算到目前为止数过的字数，还有一个boolean用来记得上一个遇到的Character是不是空格。因为Java没有元组（tuple，用来表示由异类元素组成的有序列表的结构，不需要包装对象），所以你必须创建一个新类WordCounter来把这个状态封装起来，如下所示。

代码清单7-5　用来在遍历Character流时计数的类

```
class WordCounter {
    private final int counter;
    private final boolean lastSpace;
    public WordCounter(int counter, boolean lastSpace) {
        this.counter = counter;
        this.lastSpace = lastSpace;
    }

    public WordCounter accumulate(Character c) {      ← 和迭代算法一样，accumulate方法一个个遍历Character
        if (Character.isWhitespace(c)) {
            return lastSpace ?
                   this :
                   new WordCounter(counter, true);
        } else {
            return lastSpace ?
                   new WordCounter(counter + 1, false)   ← 上一个字符是空格，而当前遍历的字符不是空格时，将单词计数器加一
                   this;
        }
    }

    public WordCounter combine(WordCounter wordCounter) {   ← 合并两个WordCounter，把其计数器加起来
        return new WordCounter(counter + wordCounter.counter,
                               wordCounter.lastSpace);   ← 仅需要计数器的总和，无需关心lastSpace
    }

    public int getCounter() {
        return counter;
    }
}
```

在这个列表中，accumulate方法定义了如何更改WordCounter的状态，或更确切地说是用哪个状态来建立新的WordCounter，因为这个类是不可变的。每次遍历到Stream中的一个新的Character时，就会调用accumulate方法。具体来说，就像代码清单7-4中的countWords-Iteratively方法一样，当上一个字符是空格，新字符不是空格时，计数器就加一。图7-7展示了accumulate方法遍历到新的Character时，WordCounter的状态转换。

调用第二个方法combine时，会对作用于Character流的两个不同子部分的两个WordCounter的部分结果进行汇总，也就是把两个WordCounter内部的计数器加起来。

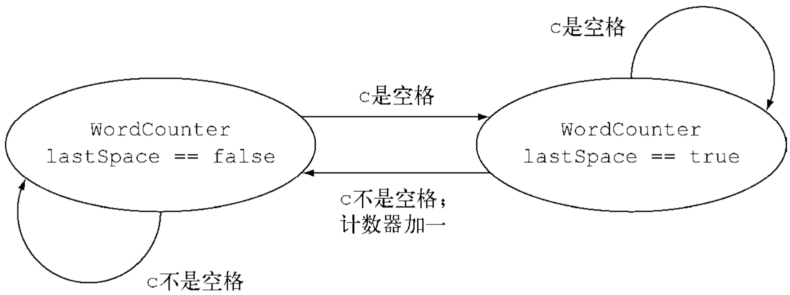


图7-7 遍历到新的Character c时WordCounter的状态转换

现在你已经写好了在WordCounter中累计字符，以及在WordCounter中把它们结合起来的逻辑，那写一个方法来归约Character流就很简单了：

```
private int countWords(Stream<Character> stream) {
    WordCounter wordCounter = stream.reduce(new WordCounter(0, true),
                                            WordCounter::accumulate,
                                            WordCounter::combine);
    return wordCounter.getCounter();
}
```

现在你就可以试一试这个方法，给它由包含但丁的《神曲》中《地狱篇》第一句的String创建的流：

```
Stream<Character> stream = IntStream.range(0, SENTENCE.length())
                                     .mapToObj(SENTENCE::charAt);
System.out.println("Found " + countWords(stream) + " words");
```

你可以和迭代版本比较一下输出：

```
Found 19 words
```

到现在为止都很好，但我们以函数式实现WordCounter的主要原因之一就是能轻松地并行处理，让我们来看看具体是如何实现的。

2. 让WordCounter并行工作

你可以尝试用并行流来加快字数统计，如下所示：

```
System.out.println("Found " + countWords(stream.parallel()) + " words");
```

不幸的是，这次的输出是：

```
Found 25 words
```

显然有什么不对，可到底是哪里不对呢？问题的根源并不难找。因为原始的String在任意位置拆分，所以有时一个词会被分为两个词，然后数了两次。这就说明，拆分流会影响结果，而把顺序流换成并行流就可能使结果出错。

如何解决这个问题呢？解决方案就是要确保String不是在随机位置拆开的，而只能在词尾拆开。要做到这一点，你必须为Character实现一个Spliterator，它只能在两个词之间拆开String（如下所示），然后由此创建并行流。

代码清单7-6　WordCounterSpliterator

```
class WordCounterSpliterator implements Spliterator<Character> {

    private final String string;
    private int currentChar = 0;

    public WordCounterSpliterator(String string) {
        this.string = string;
    }

    @Override
    public boolean tryAdvance(Consumer<? super Character> action) {
        action.accept(string.charAt(currentChar++));
        return currentChar < string.length();
    }

    @Override
    public Spliterator<Character> trySplit() {
        int currentSize = string.length() - currentChar;
        if (currentSize < 10) {
            return null;
        }
        for (int splitPos = currentSize / 2 + currentChar;
                 splitPos < string.length(); splitPos++) {
            if (Character.isWhitespace(string.charAt(splitPos))) {
                Spliterator<Character> spliterator =
                    new WordCounterSpliterator(string.substring(currentChar,
                                                                splitPos));
                currentChar = splitPos;
                return spliterator;
            }
        }
        return null;
    }

    @Override
    public long estimateSize() {
        return string.length() - currentChar;
    }

    @Override
    public int characteristics() {
        return ORDERED + SIZED + SUBSIZED + NONNULL + IMMUTABLE;
    }
}
```

处理当前字符

如果还有字符要处理，则返回**true**

将试探拆分位置设定为要解析的String的中间

返回**null**表示要解析的String已经足够小，可以顺序处理

让拆分位置前进直到下一个空格

将这个WordCounter-Spliterator的起始位置设为拆分位置

创建一个新WordCounter-Spliterator来解析String从开始到拆分位置的部分

这个Spliterator由要解析的String创建，并遍历了其中的Character，同时保存了当前正在遍历的字符位置。让我们快速回顾一下实现了Spliterator接口的WordCounterSpliterator中的各个函数。

- tryAdvance方法把String中当前位置的Character传给了Consumer，并让位置加一。作为参数传递的Consumer是一个Java内部类，在遍历流时将要处理的Character传给了

一系列要对其执行的函数。这里只有一个归约函数，即WordCounter类的accumulate方法。如果新的指针位置小于String的总长，且还有要遍历的Character，则tryAdvance返回true。

- trySplit方法是Spliterator中最重要的一个方法，因为它定义了拆分要遍历的数据结构的逻辑。就像在代码清单7-1中实现的RecursiveTask的compute方法一样（分支/合并框架的使用方式），首先要设定不再进一步拆分的下限。这里用了一个非常低的下限——10个Character，仅仅是为了保证程序会对那个比较短的String做几次拆分。在实际应用中，就像分支/合并的例子那样，你肯定要用更高的下限来避免生成太多的任务。如果剩余的Character数量低于下限，你就返回null表示无需进一步拆分。相反，如果你需要执行拆分，就把试探的拆分位置设在要解析的String块的中间。但我们没有直接使用这个拆分位置，因为要避免把词在中间断开，于是就往前找，直到找到一个空格。一旦找到了适当的拆分位置，就可以创建一个新的Spliterator来遍历从当前位置到拆分位置的子串；把当前位置this设为拆分位置，因为之前的部分将由新Spliterator来处理，最后返回。
- 还需要遍历的元素的estimatedSize就是这个Spliterator解析的String的总长度和当前遍历的位置的差。
- 最后，characteristic方法告诉框架这个Spliterator是ORDERED（顺序就是String中各个Character的次序）、SIZED（estimatedSize方法的返回值是精确的）、SUBSIZED（trySplit方法创建的其他Spliterator也有确切大小）、NONNULL（String中不能有为null的Character）和IMMUTABLE（在解析String时不能再添加Character，因为String本身是一个不可变类）的。

3. 运用**WordCounterSpliterator**

现在就可以用这个新的WordCounterSpliterator来处理并行流了，如下所示：

```
Spliterator<Character> spliterator = new WordCounterSpliterator(SENTENCE);
Stream<Character> stream = StreamSupport.stream(spliterator, true);
```

传给StreamSupport.stream工厂方法的第二个布尔参数意味着你想创建一个并行流。把这个并行流传给countWords方法：

```
System.out.println("Found " + countWords(stream) + " words");
```

可以得到意料之中的正确输出：

```
Found 19 words
```

你已经看到了Spliterator如何让你控制拆分数据结构的策略。Spliterator还有最后一个值得注意的功能，就是可以在第一次遍历、第一次拆分或第一次查询估计大小时绑定元素的数据源，而不是在创建时就绑定。这种情况下，它称为*延迟绑定*（late-binding）的Spliterator。我们专门用附录C来展示如何开发一个工具类来利用这个功能在同一个流上执行多个操作。

7.4　小结

在本章中，你了解了以下内容。

- 内部迭代让你可以并行处理一个流，而无需在代码中显式使用和协调不同的线程。
- 虽然并行处理一个流很容易，却不能保证程序在所有情况下都运行得更快。并行软件的行为和性能有时是违反直觉的，因此一定要测量，确保你并没有把程序拖得更慢。
- 像并行流那样对一个数据集并行执行操作可以提升性能，特别是要处理的元素数量庞大，或处理单个元素特别耗时的时候。
- 从性能角度来看，使用正确的数据结构，如尽可能利用原始流而不是一般化的流，几乎总是比尝试并行化某些操作更为重要。
- 分支/合并框架让你得以用递归方式将可以并行的任务拆分成更小的任务，在不同的线程上执行，然后将各个子任务的结果合并起来生成整体结果。
- `Spliterator`定义了并行流如何拆分它要遍历的数据。

Part 3 第三部分

高效 Java 8 编程

本书第三部分将探究如何结合现代程序设计方法利用Java 8的各种特性更有效地改善代码质量。

第8章会介绍如何利用Java 8的新特性及一些技巧，改进现有代码。除此之外，还会探讨一些非常重要的软件开发技术，譬如设计模式、重构、测试以及调试。

第9章中，你会了解什么是默认方法，如何以兼容的方式使用默认方法改进API，一些实用的使用模式，以及有效地利用默认方法的规则。

第10章围绕Java 8中全新引入的`java.util.Optional`类展开。`java.util.Optional`类能帮助我们设计出更优秀的API，同时降低了空指针异常发生的几率。

第11章着重介绍`CompletableFuture`类。通过`CompletableFuture`类，我们能以声明性方式描述复杂的异步计算，即并行Stream API的设计。

第12章探讨了新的Date和Time接口，这些新接口极大地优化了之前处理日期和时间时极易出错的API。

第 8 章

重构、测试和调试

8

本章内容

- 如何使用Lambda表达式重构代码
- Lambda表达式对面向对象的设计模式的影响
- Lambda表达式的测试
- 如何调试使用Lambda表达式和Stream API的代码

通过本书的前七章，我们了解了Lambda和Stream API的强大威力。你可能主要在新项目的代码中使用这些特性。如果你创建的是全新的Java项目，这是极好的时机，你可以轻装上阵，迅速地将新特性应用到项目中。然而不幸的是，大多数情况下你没有机会从头开始一个全新的项目。很多时候，你不得不面对的是用老版Java接口编写的遗留代码。

这些就是本章要讨论的内容。我们会介绍几种方法，帮助你重构代码，以适配使用Lambda表达式，让你维护的代码具备更好的可读性和灵活性。除此之外，我们还会讨论目前比较流行的几种面向对象的设计模式，包括策略模式、模板方法模式、观察者模式、责任链模式，以及工厂模式，在结合Lambda表达式之后变得更简洁的情况。最后，我们会介绍如何测试和调试使用Lambda表达式和Stream API的代码。

8.1 为改善可读性和灵活性重构代码

从本书的开篇我们就一直在强调，利用Lambda表达式，你可以写出更简洁、更灵活的代码。用“更简洁”来描述Lambda表达式是因为相较于匿名类，Lambda表达式可以帮助我们用更紧凑的方式描述程序的行为。第3章中我们也提到，如果你希望将一个既有的方法作为参数传递给另一个方法，那么方法引用无疑是我们推荐的方法，利用这种方式我们能写出非常简洁的代码。

采用Lambda表达式之后，你的代码会变得更加灵活，因为Lambda表达式鼓励大家使用第2章中介绍过的行为参数化的方式。在这种方式下，应对需求的变化时，你的代码可以依据传入的参数动态选择和执行相应的行为。

这一节，我们会将所有这些综合在一起，通过例子展示如何运用前几章介绍的Lambda表达式、方法引用以及Stream接口等特性重构遗留代码，改善程序的可读性和灵活性。

8.1.1 改善代码的可读性

改善代码的可读性到底意味着什么？我们很难定义什么是好的可读性，因为这可能非常主观。通常的理解是，“别人理解这段代码的难易程度”。改善可读性意味着你要确保你的代码能非常容易地被包括自己在内的所有人理解和维护。为了确保你的代码能被其他人理解，有几个步骤可以尝试，比如确保你的代码附有良好的文档，并严格遵守编程规范。

跟之前的版本相比较，Java 8的新特性也可以帮助提升代码的可读性：

- 使用Java 8，你可以减少冗长的代码，让代码更易于理解
- 通过方法引用和Stream API，你的代码会变得更直观

这里我们会介绍三种简单的重构，利用Lambda表达式、方法引用以及Stream改善程序代码的可读性：

- 重构代码，用Lambda表达式取代匿名类
- 用方法引用重构Lambda表达式
- 用Stream API重构命令式的数据处理

8.1.2 从匿名类到 Lambda 表达式的转换

你值得尝试的第一种重构，也是简单的方式，是将实现单一抽象方法的匿名类转换为Lambda表达式。为什么呢？前面几章的介绍应该足以说服你，因为匿名类是极其繁琐且容易出错的。采用Lambda表达式之后，你的代码会更简洁，可读性更好。比如，第3章的例子就是一个创建`Runnable`对象的匿名类，这段代码及其对应的Lambda表达式实现如下：

```
Runnable r1 = new Runnable(){        <-- 传统的方式，使用匿名类
    public void run(){
        System.out.println("Hello");
    }
};
Runnable r2 = () -> System.out.println("Hello");        <-- 新的方式，使用Lambda表达式
```

但是某些情况下，将匿名类转换为Lambda表达式可能是一个比较复杂的过程[1]。首先，匿名类和Lambda表达式中的`this`和`super`的含义是不同的。在匿名类中，`this`代表的是类自身，但是在Lambda中，它代表的是包含类。其次，匿名类可以屏蔽包含类的变量，而Lambda表达式不能（它们会导致编译错误），譬如下面这段代码：

```
int a = 10;
Runnable r1 = () -> {
    int a = 2;                <-- 编译错误！
    System.out.println(a);
};
Runnable r2 = new Runnable(){
```

① 这篇文章对转换的整个过程进行了深入细致的描述，值得一读：http://dig.cs.illinois.edu/papers/lambda-Refactoring.pdf。

8

```
    public void run(){
        int a = 2;                              // 一切正常
        System.out.println(a);
    }
};
```

最后，在涉及重载的上下文里，将匿名类转换为Lambda表达式可能导致最终的代码更加晦涩。实际上，匿名类的类型是在初始化时确定的，而Lambda的类型取决于它的上下文。通过下面这个例子，我们可以了解问题是如何发生的。我们假设你用与Runnable同样的签名声明了一个函数接口，我们称之为Task（你希望采用与你的业务模型更贴切的接口名时，就可能做这样的变更）：

```
interface Task{
    public void execute();
}
public static void doSomething(Runnable r){ r.run(); }
public static void doSomething(Task a){ a.execute(); }
```

现在，你再传递一个匿名类实现的Task，不会碰到任何问题：

```
doSomething(new Task() {
    public void execute() {
        System.out.println("Danger danger!!");
    }
});
```

但是将这种匿名类转换为Lambda表达式时，就导致了一种晦涩的方法调用，因为Runnable和Task都是合法的目标类型：

```
doSomething(() -> System.out.println("Danger danger!!"));
// 麻烦来了：doSomething(Runnable) 和 doSomething(Task) 都匹配该类型
```

你可以对Task尝试使用显式的类型转换来解决这种模棱两可的情况：

```
doSomething((Task)() -> System.out.println("Danger danger!!"));
```

但是不要因此而放弃对Lambda的尝试。好消息是，目前大多数的集成开发环境，比如NetBeans和IntelliJ都支持这种重构，它们能自动地帮你检查，避免发生这些问题。

8.1.3　从 Lambda 表达式到方法引用的转换

Lambda表达式非常适用于需要传递代码片段的场景。不过，为了改善代码的可读性，也请尽量使用方法引用。因为方法名往往能更直观地表达代码的意图。比如，第6章中我们曾经展示过下面这段代码，它的功能是按照食物的热量级别对菜肴进行分类：

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream()
        .collect(
            groupingBy(dish -> {
```

```
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT;
        }));
```

你可以将Lambda表达式的内容抽取到一个单独的方法中，将其作为参数传递给groupingBy方法。变换之后，代码变得更加简洁，程序的意图也更加清晰了：

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream().collect(groupingBy(Dish::getCaloricLevel));
```

将 Lambda 表达式抽取到一个方法内

为了实现这个方案，你还需要在Dish类中添加getCaloricLevel方法：

```
public class Dish{
    …
    public CaloricLevel getCaloricLevel(){
        if (this.getCalories() <= 400) return CaloricLevel.DIET;
        else if (this.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
    }
}
```

除此之外，我们还应该尽量考虑使用静态辅助方法，比如comparing、maxBy。这些方法设计之初就考虑了会结合方法引用一起使用。通过示例，我们看到相对于第3章中的对应代码，优化过的代码更清晰地表达了它的设计意图：


```
inventory.sort(
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
inventory.sort(comparing(Apple::getWeight));
```

你需要考虑如何实现比较算法

读起来就像问题描述，非常清晰

此外，很多通用的归约操作，比如sum、maximum，都有内建的辅助方法可以和方法引用结合使用。比如，在我们的示例代码中，使用Collectors接口可以轻松得到和或者最大值，与采用Lambada表达式和底层的归约操作比起来，这种方式要直观得多。与其编写：

```
int totalCalories =
    menu.stream().map(Dish::getCalories)
                 .reduce(0, (c1, c2) -> c1 + c2);
```

不如尝试使用内置的集合类，它能更清晰地表达问题陈述是什么。下面的代码中，我们使用了集合类summingInt（方法的名词很直观地解释了它的功能）：

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

8.1.4 从命令式的数据处理切换到 Stream

我们建议你将所有使用迭代器这种数据处理模式处理集合的代码都转换成Stream API的方式。为什么呢？Stream API能更清晰地表达数据处理管道的意图。除此之外，通过短路和延迟载入以及利用第7章介绍的现代计算机的多核架构，我们可以对Stream进行优化。

比如，下面的命令式代码使用了两种模式：筛选和抽取，这两种模式被混在了一起，这样的代码结构迫使程序员必须彻底搞清楚程序的每个细节才能理解代码的功能。此外，实现需要并行运行的程序所面对的困难也多得多（具体细节可以参考7.2节的分支/合并框架）：

```
List<String> dishNames = new ArrayList<>();
for(Dish dish: menu){
    if(dish.getCalories() > 300){
        dishNames.add(dish.getName());
    }
}
```

替代方案使用Stream API，采用这种方式编写的代码读起来更像是问题陈述，并行化也非常容易：

```
menu.parallelStream()
    .filter(d -> d.getCalories() > 300)
    .map(Dish::getName)
    .collect(toList());
```

不幸的是，将命令式的代码结构转换为Stream API的形式是个困难的任务，因为你需要考虑控制流语句，比如`break`、`continue`、`return`，并选择使用恰当的流操作。好消息是已经有一些工具可以帮助我们完成这个任务[1]。

8.1.5　增加代码的灵活性

第2章和第3章中，我们曾经介绍过Lambda表达式有利于*行为参数化*。你可以使用不同的Lambda表示不同的行为，并将它们作为参数传递给函数去处理执行。这种方式可以帮助我们淡定从容地面对需求的变化。比如，我们可以用多种方式为`Predicate`创建筛选条件，或者使用`Comparator`对多种对象进行比较。现在，我们来看看哪些模式可以马上应用到你的代码中，让你享受Lambda表达式带来的便利。

1 采用函数接口

首先，你必须意识到，没有函数接口，你就无法使用Lambda表达式。因此，你需要在代码中引入函数接口。听起来很合理,但是在什么情况下使用它们呢？这里我们介绍两种通用的模式，你可以依照这两种模式重构代码，利用Lambda表达式带来的灵活性，它们分别是：*有条件的延迟执行*和*环绕执行*。除此之外，在下一节，我们还将介绍一些基于面向对象的设计模式，比如策略模式或者模板方法，这些在使用Lambda表达式重写后会更简洁。

2. 有条件的延迟执行

我们经常看到这样的代码，控制语句被混杂在业务逻辑代码之中。典型的情况包括进行安全性检查以及日志输出。比如，下面的这段代码，它使用了Java语言内置的`Logger`类：

```
if (logger.isLoggable(Log.FINER)){
    logger.finer("Problem: " + generateDiagnostic());
}
```

① 请参考http://refactoring.info/tools/LambdaFicator/。

这段代码有什么问题吗？其实问题不少。

- 日志器的状态（它支持哪些日志等级）通过isLoggable方法暴露给了客户端代码。
- 为什么要在每次输出一条日志之前都去查询日志器对象的状态？这只能搞砸你的代码。

更好的方案是使用log方法，该方法在输出日志消息之前，会在内部检查日志对象是否已经设置为恰当的日志等级：

```
logger.log(Level.FINER, "Problem: " + generateDiagnostic());
```

这种方式更好的原因是你不再需要在代码中插入那些条件判断，与此同时日志器的状态也不再被暴露出去。不过，这段代码依旧存在一个问题。日志消息的输出与否每次都需要判断，即使你已经传递了参数，不开启日志。

这就是Lambda表达式可以施展拳脚的地方。你需要做的仅仅是延迟消息构造，如此一来，日志就只会在某些特定的情况下才开启（以此为例，当日志器的级别设置为FINER时）。显然，Java 8的API设计者们已经意识到这个问题，并由此引入了一个对log方法的重载版本，这个版本的log方法接受一个Supplier作为参数。这个替代版本的log方法的函数签名如下：

```
public void log(Level level, Supplier<String> msgSupplier)
```

你可以通过下面的方式对它进行调用：

```
logger.log(Level.FINER, () -> "Problem: " + generateDiagnostic());
```

8

如果日志器的级别设置恰当，log方法会在内部执行作为参数传递进来的Lambda表达式。这里介绍的Log方法的内部实现如下：

```
public void log(Level level, Supplier<String> msgSupplier){
    if(logger.isLoggable(level)){
        log(level, msgSupplier.get());      <-- 执行Lambda表达式
    }
}
```

从这个故事里我们学到了什么呢？如果你发现你需要频繁地从客户端代码去查询一个对象的状态（比如前文例子中的日志器的状态），只是为了传递参数、调用该对象的一个方法（比如输出一条日志），那么可以考虑实现一个新的方法，以Lambda或者方法表达式作为参数，新方法在检查完该对象的状态之后才调用原来的方法。你的代码会因此而变得更易读（结构更清晰），封装性更好（对象的状态也不会暴露给客户端代码了）。

3. 环绕执行

第3章中，我们介绍过另一种值得考虑的模式，那就是环绕执行。如果你发现虽然你的业务代码千差万别，但是它们拥有同样的准备和清理阶段，这时，你完全可以将这部分代码用Lambda实现。这种方式的好处是可以重用准备和清理阶段的逻辑，减少重复冗余的代码。下面这段代码你在第3章中已经看过，我们再回顾一次。它在打开和关闭文件时使用了同样的逻辑，但在处理文件时可以使用不同的Lambda进行参数化。

```
String oneLine =
    processFile((BufferedReader b) -> b.readLine());
String twoLines =
    processFile((BufferedReader b) -> b.readLine() + b.readLine());

public static String processFile(BufferedReaderProcessor p) throws
     IOException {
    try(BufferedReader br = new BufferedReader(new FileReader("java8inaction/
     chap8/data.txt"))){
        return p.process(br);
    }
}

public interface BufferedReaderProcessor{
    String process(BufferedReader b) throws IOException;
}
```

传入一个Lambda表达式

传入另一个Lambda表达式

将**BufferedReaderProcessor**作为执行参数传入

使用Lambda表达式的函数接口，该接口能够抛出一个**IOException**

这一优化是凭借函数式接口`BufferedReaderProcessor`达成的，通过这个接口，你可以传递各种Lamba表达式对`BufferedReader`对象进行处理。

通过这一节，你已经了解了如何通过不同方式来改善代码的可读性和灵活性。接下来，你会了解Lambada表达式如何避免常规面向对象设计中的僵化的模板代码。

8.2　使用 Lambda 重构面向对象的设计模式

新的语言特性常常让现存的编程模式或设计黯然失色。比如， Java 5中引入了`for-each`循环，由于它的稳健性和简洁性，已经替代了很多显式使用迭代器的情形。Java 7中推出的菱形操作符（<>）让大家在创建实例时无需显式使用泛型，一定程度上推动了Java程序员们采用类型接口（type interface）进行程序设计。

对设计经验的归纳总结被称为*设计模式*[①]。设计软件时，如果你愿意，可以复用这些方式方法来解决一些常见问题。这看起来像传统建筑工程师的工作方式，对典型的场景（比如悬挂桥、拱桥等）都定义有可重用的解决方案。例如，*访问者模式*常用于分离程序的算法和它的操作对象。*单例模式*一般用于限制类的实例化，仅生成一份对象。

Lambda表达式为程序员的工具箱又新添了一件利器。它们为解决传统设计模式所面对的问题提供了新的解决方案，不但如此，采用这些方案往往更高效、更简单。使用Lambda表达式后，很多现存的略显臃肿的面向对象设计模式能够用更精简的方式实现了。这一节中，我们会针对五个设计模式展开讨论，它们分别是：

- 策略模式
- 模板方法
- 观察者模式
- 责任链模式
- 工厂模式

① 参见http://c2.com/cgi/wiki?GangOfFour。

我们会展示Lambda表达式是如何另辟蹊径解决设计模式原来试图解决的问题的。

8.2.1 策略模式

策略模式代表了解决一类算法的通用解决方案，你可以在运行时选择使用哪种方案。在第2章中你已经简略地了解过这种模式了，当时我们介绍了如何使用不同的条件（比如苹果的重量，或者颜色）来筛选库存中的苹果。你可以将这一模式应用到更广泛的领域，比如使用不同的标准来验证输入的有效性，使用不同的方式来分析或者格式化输入。

策略模式包含三部分内容，如图8-1所示。

- 一个代表某个算法的接口（它是策略模式的接口）。
- 一个或多个该接口的具体实现，它们代表了算法的多种实现（比如，实体类ConcreteStrategyA或者ConcreteStrategyB）。
- 一个或多个使用策略对象的客户。

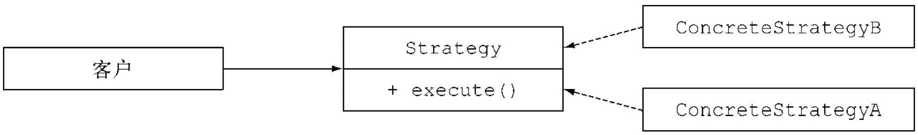


图8-1 策略模式

8

我们假设你希望验证输入的内容是否根据标准进行了恰当的格式化（比如只包含小写字母或数字）。你可以从定义一个验证文本（以String的形式表示）的接口入手：

```
public interface ValidationStrategy {
    boolean execute(String s);
}
```

其次，你定义了该接口的一个或多个具体实现：

```
public class IsAllLowerCase implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("[a-z]+");
    }
}

public class IsNumeric implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("\\d+");
    }
}
```

之后，你就可以在你的程序中使用这些略有差异的验证策略了：

```
public class Validator{
    private final ValidationStrategy strategy;

    public Validator(ValidationStrategy v){
        this.strategy = v;
```

```
    }

    public boolean validate(String s){
        return strategy.execute(s);
    }
}

Validator numericValidator = new Validator(new IsNumeric());
boolean b1 = numericValidator.validate("aaaa");            ← 返回false
Validator lowerCaseValidator = new Validator(new IsAllLowerCase ());
boolean b2 = lowerCaseValidator.validate("bbbb");          ← 返回true
```

使用Lambda表达式

到现在为止，你应该已经意识到ValidationStrategy是一个函数接口了（除此之外，它还与Predicate<String>具有同样的函数描述）。这意味着我们不需要声明新的类来实现不同的策略，通过直接传递Lambda表达式就能达到同样的目的，并且还更简洁：

```
Validator numericValidator =
    new Validator((String s) -> s.matches("[a-z]+"));     ← 直接传递Lambda表达式
boolean b1 = numericValidator.validate("aaaa");
Validator lowerCaseValidator =
    new Validator((String s) -> s.matches("\\d+"));       ← 直接传递Lambda表达式
boolean b2 = lowerCaseValidator.validate("bbbb");
```

正如你看到的，Lambda表达式避免了采用策略设计模式时僵化的模板代码。如果你仔细分析一下个中缘由，可能会发现，Lambda表达式实际已经对部分代码（或策略）进行了封装，而这就是创建策略设计模式的初衷。因此，我们强烈建议对类似的问题，你应该尽量使用Lambda表达式来解决。

8.2.2　模板方法

如果你需要采用某个算法的框架，同时又希望有一定的灵活度，能对它的某些部分进行改进，那么采用模板方法设计模式是比较通用的方案。好吧，这样讲听起来有些抽象。换句话说，模板方法模式在你“希望使用这个算法，但是需要对其中的某些行进行改进，才能达到希望的效果”时是非常有用的。

让我们从一个例子着手，看看这个模式是如何工作的。假设你需要编写一个简单的在线银行应用。通常，用户需要输入一个用户账户，之后应用才能从银行的数据库中得到用户的详细信息，最终完成一些让用户满意的操作。不同分行的在线银行应用让客户满意的方式可能还略有不同，比如给客户的账户发放红利，或者仅仅是少发送一些推广文件。你可能通过下面的抽象类方式来实现在线银行应用：

```
abstract class OnlineBanking {

    public void processCustomer(int id){
        Customer c = Database.getCustomerWithId(id);
        makeCustomerHappy(c);
    }
```

```
    abstract void makeCustomerHappy(Customer c);
}
```

processCustomer方法搭建了在线银行算法的框架：获取客户提供的ID，然后提供服务让用户满意。不同的支行可以通过继承OnlineBanking类，对该方法提供差异化的实现。

使用Lambda表达式

使用你偏爱的Lambda表达式同样也可以解决这些问题（创建算法框架，让具体的实现插入某些部分）。你想要插入的不同算法组件可以通过Lambda表达式或者方法引用的方式实现。

这里我们向processCustomer方法引入了第二个参数，它是一个Consumer<Customer>类型的参数，与前文定义的makeCustomerHappy的特征保持一致：

```
public void processCustomer(int id, Consumer<Customer> makeCustomerHappy){
    Customer c = Database.getCustomerWithId(id);
    makeCustomerHappy.accept(c);
}
```

现在，你可以很方便地通过传递Lambda表达式，直接插入不同的行为，不再需要继承OnlineBanking类了：

```
new OnlineBankingLambda().processCustomer(1337, (Customer c) ->
    System.out.println("Hello " + c.getName());
```

这是又一个例子，佐证了Lamba表达式能帮助你解决设计模式与生俱来的设计僵化问题。

8.2.3　观察者模式

观察者模式是一种比较常见的方案，某些事件发生时（比如状态转变），如果一个对象（通常我们称之为*主题*）需要自动地通知其他多个对象（称为*观察者*），就会采用该方案。创建图形用户界面（GUI）程序时，你经常会使用该设计模式。这种情况下，你会在图形用户界面组件（比如按钮）上注册一系列的观察者。如果点击按钮，观察者就会收到通知，并随即执行某个特定的行为。 但是观察者模式并不局限于图形用户界面。比如，观察者设计模式也适用于股票交易的情形，多个券商可能都希望对某一支股票价格（主题）的变动做出响应。图8-2通过UML图解释了观察者模式。

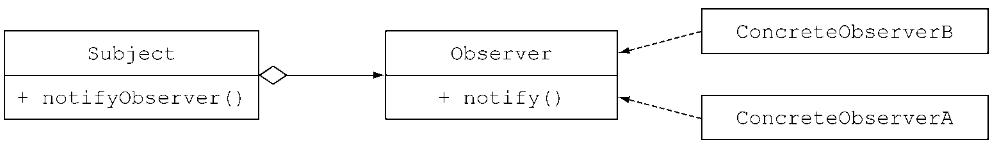


图8-2　观察者设计模式

让我们写点儿代码来看看观察者模式在实际中多么有用。你需要为Twitter这样的应用设计并实现一个定制化的通知系统。想法很简单：好几家报纸机构，比如《纽约时报》《卫报》以及《世界报》都订阅了新闻，他们希望当接收的新闻中包含他们感兴趣的关键字时，能得到特别通知。

首先，你需要一个观察者接口，它将不同的观察者聚合在一起。它仅有一个名为notify的方法，一旦接收到一条新的新闻，该方法就会被调用：

```
interface Observer {
    void notify(String tweet);
}
```

现在，你可以声明不同的观察者（比如，这里是三家不同的报纸机构），依据新闻中不同的关键字分别定义不同的行为：

```
class NYTimes implements Observer{
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("money")){
            System.out.println("Breaking news in NY! " + tweet);
        }
    }
}
class Guardian implements Observer{
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("queen")){
            System.out.println("Yet another news in London... " + tweet);
        }
    }
}
class LeMonde implements Observer{
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("wine")){
            System.out.println("Today cheese, wine and news! " + tweet);
        }
    }
}
```

你还遗漏了最重要的部分：Subject！让我们为它定义一个接口：

```
interface Subject{
    void registerObserver(Observer o);
    void notifyObservers(String tweet);
}
```

Subject使用registerObserver方法可以注册一个新的观察者，使用notifyObservers方法通知它的观察者一个新闻的到来。让我们更进一步，实现Feed类：

```
class Feed implements Subject{

    private final List<Observer> observers = new ArrayList<>();

    public void registerObserver(Observer o) {
        this.observers.add(o);
    }

    public void notifyObservers(String tweet) {
        observers.forEach(o -> o.notify(tweet));
    }
}
```

这是一个非常直观的实现：Feed类在内部维护了一个观察者列表，一条新闻到达时，它就进行通知。

```
Feed f = new Feed();
f.registerObserver(new NYTimes());
f.registerObserver(new Guardian());
f.registerObserver(new LeMonde());
f.notifyObservers("The queen said her favourite book is Java 8 in Action!");
```

毫不意外，《卫报》会特别关注这条新闻！

使用Lambda表达式

你可能会疑惑Lambda表达式在观察者设计模式中如何发挥它的作用。不知道你有没有注意到，Observer接口的所有实现类都提供了一个方法：notify。新闻到达时，它们都只是对同一段代码封装执行。Lambda表达式的设计初衷就是要消除这样的僵化代码。使用Lambda表达式后，你无需显式地实例化三个观察者对象，直接传递Lambda表达式表示需要执行的行为即可：

```
f.registerObserver((String tweet) -> {
    if(tweet != null && tweet.contains("money")){
        System.out.println("Breaking news in NY! " + tweet);
    }
});

f.registerObserver((String tweet) -> {
    if(tweet != null && tweet.contains("queen")){
        System.out.println("Yet another news in London... " + tweet);
    }
});
```

那么，是否我们随时随地都可以使用Lambda表达式呢？答案是否定的！我们前文介绍的例子中，Lambda适配得很好，那是因为需要执行的动作都很简单，因此才能很方便地消除僵化代码。但是，观察者的逻辑有可能十分复杂，它们可能还持有状态，抑或定义了多个方法，诸如此类。在这些情形下，你还是应该继续使用类的方式。

8.2.4 责任链模式

责任链模式是一种创建处理对象序列（比如操作序列）的通用方案。一个处理对象可能需要在完成一些工作之后，将结果传递给另一个对象，这个对象接着做一些工作，再转交给下一个处理对象，以此类推。

通常，这种模式是通过定义一个代表处理对象的抽象类来实现的，在抽象类中会定义一个字段来记录后续对象。一旦对象完成它的工作，处理对象就会将它的工作转交给它的后继。代码中，这段逻辑看起来是下面这样：

```
public abstract class ProcessingObject<T> {

    protected ProcessingObject<T> successor;
    public void setSuccessor(ProcessingObject<T> successor){
        this.successor = successor;
```

```
    }

    public T handle(T input){
        T r = handleWork(input);
        if(successor != null){
            return successor.handle(r);
        }
        return r;
    }

    abstract protected T handleWork(T input);
}
```

图8-3以UML的方式阐释了责任链模式。

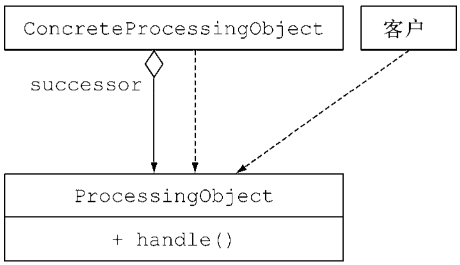


图8-3　责任链设计模式

可能你已经注意到，这就是8.2.2节介绍的模板方法设计模式。`handle`方法提供了如何进行工作处理的框架。不同的处理对象可以通过继承`ProcessingObject`类，提供`handleWork`方法来进行创建。

下面让我们看看如何使用该设计模式。你可以创建两个处理对象，它们的功能是进行一些文本处理工作。

```
public class HeaderTextProcessing extends ProcessingObject<String> {
    public String handleWork(String text){
        return "From Raoul, Mario and Alan: " + text;
    }
}

public class SpellCheckerProcessing extends ProcessingObject<String> {
    public String handleWork(String text){
        return text.replaceAll("labda", "lambda");    // 糟糕，我们漏掉了Lambda中的m字符
    }
}
```

现在你就可以将这两个处理对象结合起来，构造一个操作序列！

```
ProcessingObject<String> p1 = new HeaderTextProcessing();
ProcessingObject<String> p2 = new SpellCheckerProcessing();

p1.setSuccessor(p2);    // 将两个处理对象链接起来

String result = p1.handle("Aren't labdas really sexy?!!");
System.out.println(result);    // 打印输出“From Raoul, Mario and Alan: Aren't lambdas really sexy?!!”
```

使用Lambda表达式

稍等！这个模式看起来像是在链接（也即是构造）函数。第3章中我们探讨过如何构造Lambda表达式。你可以将处理对象作为函数的一个实例，或者更确切地说作为`UnaryOperator<String>`的一个实例。为了链接这些函数，你需要使用`andThen`方法对其进行构造。

```
UnaryOperator<String> headerProcessing =
    (String text) -> "From Raoul, Mario and Alan: " + text;    // 第一个处理对象

UnaryOperator<String> spellCheckerProcessing =
    (String text) -> text.replaceAll("labda", "lambda");    // 第二个处理对象

Function<String, String> pipeline =
    headerProcessing.andThen(spellCheckerProcessing);    // 将两个方法结合起来，结果就是一个操作链

String result = pipeline.apply("Aren't labdas really sexy?!!")
```

8.2.5 工厂模式

使用工厂模式，你无需向客户暴露实例化的逻辑就能完成对象的创建。比如，我们假定你为一家银行工作，他们需要一种方式创建不同的金融产品：贷款、期权、股票，等等。

通常，你会创建一个工厂类，它包含一个负责实现不同对象的方法，如下所示：

```
public class ProductFactory {
    public static Product createProduct(String name){
        switch(name){
            case "loan": return new Loan();
            case "stock": return new Stock();
            case "bond": return new Bond();
            default: throw new RuntimeException("No such product " + name);
        }
    }
}
```

这里贷款（`Loan`）、股票（`Stock`）和债券（`Bond`）都是产品（`Product`）的子类。`createProduct`方法可以通过附加的逻辑来设置每个创建的产品。但是带来的好处也显而易见，你在创建对象时不用再担心会将构造函数或者配置暴露给客户，这使得客户创建产品时更加简单：

```
Product p = ProductFactory.createProduct("loan");
```

使用Lambda表达式

第3章中，我们已经知道可以像引用方法一样引用构造函数。比如，下面就是一个引用贷款（`Loan`）构造函数的示例：

```
Supplier<Product> loanSupplier = Loan::new;
Loan loan = loanSupplier.get();
```

通过这种方式，你可以重构之前的代码，创建一个`Map`，将产品名映射到对应的构造函数：

8

```
final static Map<String, Supplier<Product>> map = new HashMap<>();
static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

现在，你可以像之前使用工厂设计模式那样，利用这个Map来实例化不同的产品。

```
public static Product createProduct(String name){
    Supplier<Product> p = map.get(name);
    if(p != null) return p.get();
    throw new IllegalArgumentException("No such product " + name);
}
```

这是个全新的尝试，它使用Java 8中的新特性达到了传统工厂模式同样的效果。但是，如果工厂方法createProduct需要接收多个传递给产品构造方法的参数，这种方式的扩展性不是很好。你不得不提供不同的函数接口，无法采用之前统一使用一个简单接口的方式。

比如，我们假设你希望保存具有三个参数（两个参数为Integer类型，一个参数为String类型）的构造函数；为了完成这个任务，你需要创建一个特殊的函数接口TriFunction。最终的结果是Map变得更加复杂。

```
public interface TriFunction<T, U, V, R>{
    R apply(T t, U u, V v);
}
Map<String, TriFunction<Integer, Integer, String, Product>> map
    = new HashMap<>();
```

你已经了解了如何使用Lambda表达式编写和重构代码。接下来，我们会介绍如何确保新编写代码的正确性。

8.3　测试 Lambda 表达式

现在你的代码中已经充溢着Lambda表达式，看起来不错，也很简洁。但是，大多数时候，我们受雇进行的程序开发工作的要求并不是编写优美的代码，而是编写正确的代码。

通常而言，好的软件工程实践一定少不了单元测试，借此保证程序的行为与预期一致。你编写测试用例，通过这些测试用例确保你代码中的每个组成部分都实现预期的结果。比如，图形应用的一个简单的Point类，可以定义如下：

```
public class Point{
    private final int x;
    private final int y;

    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
```

```
    public int getY() { return y; }
    public Point moveRightBy(int x){
        return new Point(this.x + x, this.y);
    }
}
```

下面的单元测试会检查moveRightBy方法的行为是否与预期一致：

```
@Test
public void testMoveRightBy() throws Exception {
    Point p1 = new Point(5, 5);
    Point p2 = p1.moveRightBy(10);

    assertEquals(15, p2.getX());
    assertEquals(5, p2.getY());
}
```

8.3.1 测试可见 Lambda 函数的行为

由于moveRightBy方法声明为public，测试工作变得相对容易。你可以在用例内部完成测试。但是Lambda并无函数名（毕竟它们都是匿名函数），因此要对你代码中的Lambda函数进行测试实际上比较困难，因为你无法通过函数名的方式调用它们。

有些时候，你可以借助某个字段访问Lambda函数，这种情况，你可以利用这些字段，通过它们对封装在Lambda函数内的逻辑进行测试。比如，我们假设你在Point类中添加了静态字段compareByXAndThenY，通过该字段，使用方法引用你可以访问Comparator对象：


```
public class Point{
    public final static Comparator<Point> compareByXAndThenY =
        comparing(Point::getX).thenComparing(Point::getY);
    …
}
```

还记得吗，Lambda表达式会生成函数接口的一个实例。由此，你可以测试该实例的行为。这个例子中，我们可以使用不同的参数，对Comparator对象类型实例compareByXAndThenY的compare方法进行调用，验证它们的行为是否符合预期：

```
@Test
public void testComparingTwoPoints() throws Exception {
    Point p1 = new Point(10, 15);
    Point p2 = new Point(10, 20);
    int result = Point.compareByXAndThenY.compare(p1 , p2);
    assertEquals(-1, result);
}
```

8.3.2 测试使用 Lambda 的方法的行为

但是Lambda的初衷是将一部分逻辑封装起来给另一个方法使用。从这个角度出发，你不应该将Lambda表达式声明为public，它们仅是具体的实现细节。相反，我们需要对使用Lambda表达

式的方法进行测试。比如下面这个方法moveAllPointsRightBy：

```
public static List<Point> moveAllPointsRightBy(List<Point> points, int x){
    return points.stream()
                 .map(p -> new Point(p.getX() + x, p.getY()))
                 .collect(toList());
}
```

我们没必要对Lambda表达式p -> new Point(p.getX() + x,p.getY())进行测试，它只是moveAllPointsRightBy内部的实现细节。我们更应该关注的是方法moveAllPointsRightBy的行为：

```
@Test
public void testMoveAllPointsRightBy() throws Exception{
    List<Point> points =
        Arrays.asList(new Point(5, 5), new Point(10, 5));
    List<Point> expectedPoints =
        Arrays.asList(new Point(15, 5), new Point(20, 5));

    List<Point> newPoints = Point.moveAllPointsRightBy(points, 10);
    assertEquals(expectedPoints, newPoints);
}
```

注意，上面的单元测试中，Point类恰当地实现equals方法非常重要，否则该测试的结果就取决于Object类的默认实现。

8.3.3　将复杂的 Lambda 表达式分到不同的方法

可能你会碰到非常复杂的Lambda表达式，包含大量的业务逻辑，比如需要处理复杂情况的定价算法。你无法在测试程序中引用Lambda表达式，这种情况该如何处理呢？一种策略是将Lambda表达式转换为方法引用（这时你往往需要声明一个新的常规方法），我们在8.1.3节详细讨论过这种情况。这之后，你可以用常规的方式对新的方法进行测试。

8.3.4　高阶函数的测试

接受函数作为参数的方法或者返回一个函数的方法（所谓的“高阶函数”，higher-order function，我们在第14章会深入展开介绍）更难测试。如果一个方法接受Lambda表达式作为参数，你可以采用的一个方案是使用不同的Lambda表达式对它进行测试。比如，你可以使用不同的谓词对第2章中创建的filter方法进行测试。

```
@Test
public void testFilter() throws Exception{
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
    List<Integer> even = filter(numbers, i -> i % 2 == 0);
    List<Integer> smallerThanThree = filter(numbers, i -> i < 3);
    assertEquals(Arrays.asList(2, 4), even);
    assertEquals(Arrays.asList(1, 2), smallerThanThree);
}
```

如果被测试方法的返回值是另一个方法，该如何处理呢？你可以仿照我们之前处理Comparator的方法，把它当成一个函数接口，对它的功能进行测试。

然而，事情可能不会一帆风顺，你的测试可能会返回错误，报告说你使用Lambda表达式的方式不对。因此，我们现在进入调试的环节。

8.4　调试

调试有问题的代码时，程序员的兵器库里有两大老式武器，分别是：

- 查看栈跟踪
- 输出日志

8.4.1　查看栈跟踪

你的程序突然停止运行（比如突然抛出一个异常），这时你首先要调查程序在什么地方发生了异常以及为什么会发生该异常。这时栈帧就非常有用。程序的每次方法调用都会产生相应的调用信息，包括程序中方法调用的位置、该方法调用使用的参数、被调用方法的本地变量。这些信息被保存在栈帧上。

程序失败时，你会得到它的*栈跟踪*，通过一个又一个栈帧，你可以了解程序失败时的概略信息。换句话说，通过这些你能得到程序失败时的方法调用列表。这些方法调用列表最终会帮助你发现问题出现的原因。

8

Lambda表达式和栈跟踪

不幸的是，由于Lambda表达式没有名字，它的栈跟踪可能很难分析。在下面这段简单的代码中，我们刻意地引入了一些错误：

```
import java.util.*;

public class Debugging{
    public static void main(String[] args) {
        List<Point> points = Arrays.asList(new Point(12, 2), null);
        points.stream().map(p -> p.getX()).forEach(System.out::println);
    }
}
```

运行这段代码会产生下面的栈跟踪：

```
Exception in thread "main" java.lang.NullPointerException
    at Debugging.lambda$main$0(Debugging.java:6)          <-- 这行中的$0是什么意思？
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
      .java:193)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators
      .java:948)
…
```

讨厌！发生了什么？这段程序当然会失败，因为Points列表的第二个元素是空（null）。

这时你的程序实际是在试图处理一个空引用。由于Stream流水线发生了错误，构成Stream流水线的整个方法调用序列都暴露在你面前了。不过，你留意到了吗？栈跟踪中还包含下面这样类似加密的内容：

```
at Debugging.lambda$main$0(Debugging.java:6)
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
```

这些表示错误发生在Lambda表达式内部。由于Lambda表达式没有名字，所以编译器只能为它们指定一个名字。这个例子中，它的名字是`lambda$main$0`，看起来非常不直观。如果你使用了大量的类，其中又包含多个Lambda表达式，这就成了一个非常头痛的问题。

即使你使用了方法引用，还是有可能出现栈无法显示你使用的方法名的情况。将之前的Lambda表达式`p-> p.getX()`替换为方法引用`reference Point::getX`也会产生难于分析的栈跟踪：

```
points.stream().map(Point::getX).forEach(System.out::println);

Exception in thread "main" java.lang.NullPointerException
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
       .java:193)
…
```

这一行表示什么呢？

注意，如果方法引用指向的是同一个类中声明的方法，那么它的名称是可以在栈跟踪中显示的。比如，下面这个例子：

```
import java.util.*;

public class Debugging{
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3);
        numbers.stream().map(Debugging::divideByZero).forEach(System
            .out::println);
    }

    public static int divideByZero(int n){
        return n / 0;
    }
}
```

方法`divideByZero`在栈跟踪中就正确地显示了：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Debugging.divideByZero(Debugging.java:10)
    at Debugging$$Lambda$1/999966131.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
       .java:193)
…
```

divideByZero 正确地输出到栈跟踪中

总的来说，我们需要特别注意，涉及Lambda表达式的栈跟踪可能非常难理解。这是Java编译器未来版本可以改进的一个方面。

8.4.2 使用日志调试

假设你试图对流操作中的流水线进行调试，该从何入手呢？你可以像下面的例子那样，使用forEach将流操作的结果日志输出到屏幕上或者记录到日志文件中：

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);

numbers.stream()
       .map(x -> x + 17)
       .filter(x -> x % 2 == 0)
       .limit(3)
       .forEach(System.out::println);
```

这段代码的输出如下：

```
20
22
```

不幸的是，一旦调用forEach，整个流就会恢复运行。到底哪种方式能更有效地帮助我们理解Stream流水线中的每个操作（比如map、filter、limit）产生的输出？

这就是流操作方法peek大显身手的时候。peek的设计初衷就是在流的每个元素恢复运行之前，插入执行一个动作。但是它不像forEach那样恢复整个流的运行，而是在一个元素上完成操作之后，它只会将操作顺承到流水线中的下一个操作。图8-4解释了peek的操作流程。下面的这段代码中，我们使用peek输出了Stream流水线操作之前和操作之后的中间值：


```
List<Integer> result =
  numbers.stream()
         .peek(x -> System.out.println("from stream: " + x))
         .map(x -> x + 17)
         .peek(x -> System.out.println("after map: " + x))
         .filter(x -> x % 2 == 0)
         .peek(x -> System.out.println("after filter: " + x))
         .limit(3)
         .peek(x -> System.out.println("after limit: " + x))
         .collect(toList());
```

输出来自数据源的当前元素值

输出map操作的结果

输出经过filter操作之后，剩下的元素个数

输出经过limit操作之后，剩下的元素个数

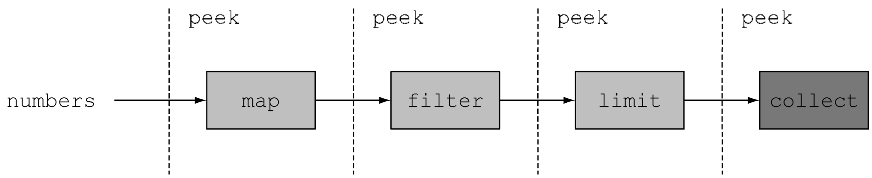


图8-4 使用peek查看Stream流水线中的数据流的值

通过peek操作我们能清楚地了解流水线操作中每一步的输出结果：

```
from stream: 2
after map: 19
```

```
from stream: 3
after map: 20
after filter: 20
after limit: 20
from stream: 4
after map: 21
from stream: 5
after map: 22
after filter: 22
after limit: 22
```

8.5　小结

下面回顾一下这一章的主要内容。

- Lambda表达式能提升代码的可读性和灵活性。
- 如果你的代码中使用了匿名类，尽量用Lambda表达式替换它们，但是要注意二者间语义的微妙差别，比如关键字`this`，以及变量隐藏。
- 跟Lambda表达式比起来，方法引用的可读性更好 。
- 尽量使用Stream API替换迭代式的集合处理。
- Lambda表达式有助于避免使用面向对象设计模式时容易出现的僵化的模板代码，典型的比如策略模式、模板方法、观察者模式、责任链模式，以及工厂模式。
- 即使采用了Lambda表达式，也同样可以进行单元测试，但是通常你应该关注使用了Lambda表达式的方法的行为。
- 尽量将复杂的Lambda表达式抽象到普通方法中。
- Lambda表达式会让栈跟踪的分析变得更为复杂。
- 流提供的`peek`方法在分析Stream流水线时，能将中间变量的值输出到日志中，是非常有用的工具。

第 9 章 默认方法

本章内容

- 什么是默认方法
- 如何以一种兼容的方式改进API
- 默认方法的使用模式
- 解析规则

传统上，Java程序的接口是将相关方法按照约定组合到一起的方式。实现接口的类必须为接口中定义的每个方法提供一个实现，或者从父类中继承它的实现。但是，一旦类库的设计者需要更新接口，向其中加入新的方法，这种方式就会出现问题。现实情况是，现存的实体类往往不在接口设计者的控制范围之内，这些实体类为了适配新的接口约定也需要进行修改。由于Java 8的API在现存的接口上引入了非常多的新方法，这种变化带来的问题也愈加严重，一个例子就是前几章中使用过的List接口上的sort方法。想象一下其他备选集合框架的维护人员会多么抓狂吧，像Guava和Apache Commons这样的框架现在都需要修改实现了List接口的所有类，为其添加sort方法的实现。

且慢，其实你不必惊慌。Java 8为了解决这一问题引入了一种新的机制。Java 8中的接口现在支持在声明方法的同时提供实现，这听起来让人惊讶！通过两种方式可以完成这种操作。其一，Java 8允许在接口内声明*静态方法*。其二，Java 8引入了一个新功能，叫*默认方法*，通过默认方法你可以指定接口方法的默认实现。换句话说，接口能提供方法的具体实现。因此，实现接口的类如果不显式地提供该方法的具体实现，就会自动继承默认的实现。这种机制可以使你平滑地进行接口的优化和演进。实际上，到目前为止你已经使用了多个默认方法。两个例子就是你前面已经见过的List接口中的sort，以及Collection接口中的stream。

第1章中我们看到的List接口中的sort方法是Java 8中全新的方法，它的定义如下：

```
default void sort(Comparator<? super E> c){
    Collections.sort(this, c);
}
```

请注意返回类型之前的新default修饰符。通过它，我们能够知道一个方法是否为默认方法。这里sort方法调用了Collections.sort方法进行排序操作。由于有了这个新的方法，我们现

在可以直接通过调用sort，对列表中的元素进行排序。

```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 6);
numbers.sort(Comparator.naturalOrder());
```

sort是List接口的默认方法

不过除此之外，这段代码中还有些其他的新东西。注意到了吗，我们调用了Comparator.naturalOrder方法。这是Comparator接口的一个全新的静态方法，它返回一个Comparator对象，并按自然序列对其中的元素进行排序（即标准的字母数字方式排序）。

第4章中你看到的Collection中的stream方法的定义如下：

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

我们在之前的几章中大量使用了该方法来处理集合，这里stream方法中调用了SteamSupport.stream方法来返回一个流。你注意到stream方法的主体是如何调用spliterator方法的了吗？它也是Collection接口的一个默认方法。

喔噢！这些接口现在看起来像抽象类了吧？是，也不是。它们有一些本质的区别，我们在这一章中会针对性地进行讨论。但更重要的是，你为什么要在乎默认方法？默认方法的主要目标用户是类库的设计者啊。正如我们后面所解释的，默认方法的引入就是为了以兼容的方式解决像Java API这样的类库的演进问题的，如图9-1所示。

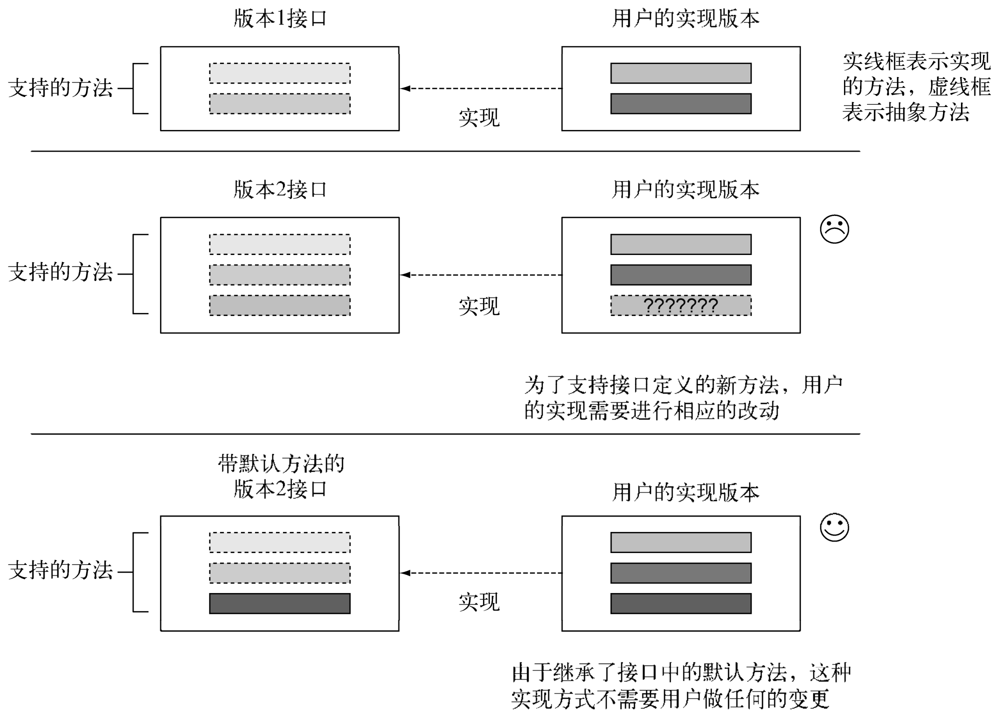


图9-1　向接口添加方法

简而言之，向接口添加方法是诸多问题的罪恶之源；一旦接口发生变化，实现这些接口的类往往也需要更新，提供新添方法的实现才能适配接口的变化。如果你对接口以及它所有相关的实现有完全的控制，这可能不是个大问题。但是这种情况是极少的。这就是引入默认方法的目的：它让类可以自动地继承接口的一个默认实现。

因此，如果你是个类库的设计者，这一章的内容对你而言会十分重要，因为默认方法为接口的演进提供了一种平滑的方式，你的改动将不会导致已有代码的修改。此外，正如我们后文会介绍的，默认方法为方法的多继承提供了一种更灵活的机制，可以帮助你更好地规划你的代码结构：类可以从多个接口继承默认方法。因此，即使你并非类库的设计者，也能在其中发现感兴趣的东西。

静态方法及接口

同时定义接口以及工具辅助类（companion class）是Java语言常用的一种模式，工具类定义了与接口实例协作的很多静态方法。比如，`Collections`就是处理`Collection`对象的辅助类。由于静态方法可以存在于接口内部，你代码中的这些辅助类就没有了存在的必要，你可以把这些静态方法转移到接口内部。为了保持后向的兼容性，这些类依然会存在于Java应用程序的接口之中。

本章的结构如下。首先，我们会跟你一起剖析一个API演化的用例，探讨由此引发的各种问题。紧接着我们会解释什么是默认方法，以及它们在这个用例中如何解决相应的问题。之后，我们会展示如何创建自己的默认方法，构造Java语言中的多继承。最后，我们会讨论一个类在使用一个签名同时继承多个默认方法时，Java编译器是如何解决可能的二义性（模糊性）问题的。

9

9.1　不断演进的 API

为了理解为什么一旦API发布之后，它的演进就变得非常困难，我们假设你是一个流行Java绘图库的设计者（为了说明本节的内容，我们做了这样的假想）。你的库中包含了一个`Resizable`接口，它定义了一个简单的可缩放形状必须支持的很多方法，比如：`setHeight`、`setWidth`、`getHeight`、`getWidth`以及`setAbsoluteSize`。此外，你还提供了几个额外的实现（out-of-box implementation），如正方形、长方形。由于你的库非常流行，你的一些用户使用`Resizable`接口创建了他们自己感兴趣的实现，比如椭圆。

发布API几个月之后，你突然意识到`Resizable`接口遗漏了一些功能。比如，如果接口提供一个`setRelativeSize`方法，可以接受参数实现对形状的大小进行调整，那么接口的易用性会更好。你会说这看起来很容易啊：为`Resizable`接口添加`setRelativeSize`方法，再更新`Square`和`Rectangle`的实现就好了。不过，事情并非如此简单！你要考虑已经使用了你接口的用户，他们已经按照自身的需求实现了`Resizable`接口，他们该如何应对这样的变更呢？非常不幸，你无

法访问，也无法改动他们实现了Resizable接口的类。这也是Java库的设计者需要改进Java API时面对的问题。让我们以一个具体的实例为例，深入探讨修改一个已发布接口的种种后果。

9.1.1　初始版本的 API

Resizable接口的最初版本提供了下面这些方法：

```
public interface Resizable extends Drawable{
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
}
```

用户实现

你的一位铁杆用户根据自身的需求实现了Resizable接口，创建了Ellipse类：

```
public class Ellipse implements Resizable {
    …
}
```

他实现了一个处理各种Resizable形状（包括Ellipse）的游戏：

```
public class Game{
    public static void main(String...args){
        List<Resizable> resizableShapes =
            Arrays.asList(new Square(), new Rectangle(), new Ellipse());   <-- 可以调整大小的形状列表
        Utils.paint(resizableShapes);
    }
}
public class Utils{
    public static void paint(List<Resizable> l){
        l.forEach(r -> {
                            r.setAbsoluteSize(42, 42);   <-- 调用每个形状自己的setAbsoluteSize方法
                            r.draw();
                        });
    }
}
```

9.1.2　第二版 API

库上线使用几个月之后，你收到很多请求，要求你更新Resizable的实现，让Square、Rectangle以及其他的形状都能支持setRelativeSize方法。为了满足这些新的需求，你发布了第二版API，具体如图9-2所示。

```
public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
```

```
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    void setRelativeSize(int wFactor, int hFactor);
}
```

第二版API添加了一个新方法

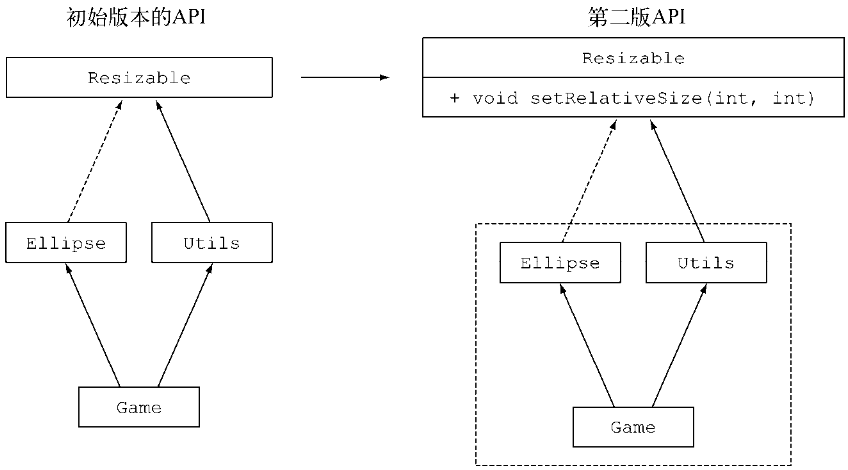


图9-2　为Resizable接口添加新方法改进API。再次编译应用时会遭遇错误，因为它依赖的Resizable接口发生了变化

用户面临的窘境

对Resizable接口的更新导致了一系列的问题。首先，接口现在要求它所有的实现类添加setRelativeSize方法的实现。但是用户最初实现的Ellipse类并未包含setRelativeSize方法。向接口添加新方法是二进制兼容的，这意味着如果不重新编译该类，即使不实现新的方法，现有类的实现依旧可以运行。不过，用户可能修改他的游戏，在他的Utils.paint方法中调用setRelativeSize方法，因为paint方法接受一个Resizable对象列表作为参数。如果传递的是一个Ellipse对象，程序就会抛出一个运行时错误，因为它并未实现setRelativeSize方法：

```
Exception in thread "main" java.lang.AbstractMethodError:
    lambdasinaction.chap9.Ellipse.setRelativeSize(II)V
```

其次，如果用户试图重新编译整个应用（包括Ellipse类），他会遭遇下面的编译错误：

```
lambdasinaction/chap9/Ellipse.java:6: error: Ellipse is not abstract and does
    not override abstract method setRelativeSize(int,int) in Resizable
```

最后，更新已发布API会导致后向兼容性问题。这就是为什么对现存API的演进，比如官方发布的Java Collection API，会给用户带来麻烦。当然，还有其他方式能够实现对API的改进，但是都不是明智的选择。比如，你可以为你的API创建不同的发布版本，同时维护老版本和新版本，但这是非常费时费力的，原因如下。其一，这增加了你作为类库的设计者维护类库的复杂度。其次，类库的用户不得不同时使用一套代码的两个版本，而这会增大内存的消耗，延长程序的载入时间，因为这种方式下项目使用的类文件数量更多了。

这就是默认方法试图解决的问题。它让类库的设计者放心地改进应用程序接口，无需担忧对遗留代码的影响，这是因为实现更新接口的类现在会自动继承一个默认的方法实现。

不同类型的兼容性：二进制、源代码和函数行为

变更对Java程序的影响大体可以分成三种类型的兼容性，分别是：二进制级的兼容、源代码级的兼容，以及函数行为的兼容。[1]刚才我们看到，向接口添加新方法是二进制级的兼容，但最终编译实现接口的类时却会发生编译错误。了解不同类型兼容性的特性是非常有益的，下面我们会深入介绍这部分的内容。

二进制级的兼容性表示现有的二进制执行文件能无缝持续链接（包括验证、准备和解析）和运行。比如，为接口添加一个方法就是二进制级的兼容，这种方式下，如果新添加的方法不被调用，接口已经实现的方法可以继续运行，不会出现错误。

简单地说，源代码级的兼容性表示引入变化之后，现有的程序依然能成功编译通过。比如，向接口添加新的方法就不是源码级的兼容，因为遗留代码并没有实现新引入的方法，所以它们无法顺利通过编译。

最后，函数行为的兼容性表示变更发生之后，程序接受同样的输入能得到同样的结果。比如，为接口添加新的方法就是函数行为兼容的，因为新添加的方法在程序中并未被调用（抑或该接口在实现中被覆盖了）。

9.2　概述默认方法

经过前述的介绍，我们已经了解了向已发布的API添加方法，对现存代码实现会造成多大的损害。默认方法是Java 8中引入的一个新特性，希望能借此以兼容的方式改进API。现在，接口包含的方法签名在它的实现类中也可以不提供实现。那么，谁来具体实现这些方法呢？实际上，缺失的方法实现会作为接口的一部分由实现类继承（所以命名为默认实现），而无需由实现类提供。

那么，我们该如何辨识哪些是默认方法呢？其实非常简单。默认方法由`default`修饰符修饰，并像类中声明的其他方法一样包含方法体。比如，你可以像下面这样在集合库中定义一个名为`Sized`的接口，在其中定义一个抽象方法`size`，以及一个默认方法`isEmpty`：

```
public interface Sized {
    int size();
    default boolean isEmpty() {      <-- 默认方法
        return size() == 0;
    }
}
```

这样任何一个实现了`Sized`接口的类都会自动继承`isEmpty`的实现。因此，向提供了默认实现的接口添加方法就不是源码兼容的。

[1] 参见https://blogs.oracle.com/darcy/entry/kinds_of_compatibility。

现在，我们回顾一下最初的例子，那个Java画图类库和你的游戏程序。具体来说，为了以兼容的方式改进这个库（即使用该库的用户不需要修改他们实现了`Resizable`的类），可以使用默认方法，提供`setRelativeSize`的默认实现：

```
default void setRelativeSize(int wFactor, int hFactor){
    setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
}
```

由于接口现在可以提供带实现的方法，是否这意味着Java已经在某种程度上实现了多继承？如果实现类也实现了同样的方法，这时会发生什么情况？默认方法会被覆盖吗？现在暂时无需担心这些，Java 8中已经定义了一些规则和机制来处理这些问题。详细的内容，我们会在9.5节进行介绍。

你可能已经猜到，默认方法在Java 8的API中已经大量地使用了。本章已经介绍过我们前一章中大量使用的`Collection`接口的`stream`方法就是默认方法。`List`接口的`sort`方法也是默认方法。第3章介绍的很多函数式接口，比如`Predicate`、`Function`以及`Comparator`也引入了新的默认方法，比如`Predicate.and`或者`Function.andThen`（记住，函数式接口只包含一个抽象方法，默认方法是种非抽象方法）。

Java 8中的抽象类和抽象接口

那么抽象类和抽象接口之间的区别是什么呢？它们不都能包含抽象方法和包含方法体的实现吗？

首先，一个类只能继承一个抽象类，但是一个类可以实现多个接口。

其次，一个抽象类可以通过实例变量（字段）保存一个通用状态，而接口是不能有实例变量的。

请应用你掌握的默认方法的知识，回答一下测验9.1的问题。

测验9.1：`removeIf`

这个测验里，假设你是Java语言和API的一个负责人。你收到了关于`removeIf`方法的很多请求，希望能为`ArrayList`、`TreeSet`、`LinkedList`以及其他集合类型添加`removeIf`方法。`removeIf`方法的功能是删除满足给定谓词的所有元素。你的任务是找到添加这个新方法、优化Collection API的最佳途径。

答案：改进Collection API破坏性最大的方式是什么？你可以把`removeIf`的实现直接复制到Collection API的每个实体类中，但这种做法实际是在对Java界的犯罪。还有其他的方式吗？你知道吗，所有的`Collection`类都实现了一个名为`java.util.Collection`的接口。太好了，那么我们可以在这里添加一个方法？是的！你只需要牢记，默认方法是一种以源码兼容方式向接口内添加实现的方法。这样实现`Collction`的所有类（包括并不隶属Collection API的用户扩展类）都能使用`removeIf`的默认实现。`removeIf`的代码实现如下（它实际就是Java 8

Collection API的实现)。它是Collection接口的一个默认方法：

```
default boolean removeIf(Predicate<? super E> filter) {
    boolean removed = false;
    Iterator<E> each = iterator();
    while(each.hasNext()) {
        if(filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

9.3　默认方法的使用模式

现在你已经了解了默认方法怎样以兼容的方式演进库函数了。除了这种用例，还有其他场景也能利用这个新特性吗？当然有，你可以创建自己的接口，并为其提供默认方法。这一节中，我们会介绍使用默认方法的两种用例：可选方法和行为的多继承。

9.3.1　可选方法

你很可能也碰到过这种情况，类实现了接口，不过却刻意地将一些方法的实现留白。我们以Iterator接口为例来说。Iterator接口定义了hasNext、next，还定义了remove方法。Java 8之前，由于用户通常不会使用该方法，remove方法常被忽略。因此，实现Interator接口的类通常会为remove方法放置一个空的实现，这些都是些毫无用处的模板代码。

采用默认方法之后，你可以为这种类型的方法提供一个默认的实现，这样实体类就无需在自己的实现中显式地提供一个空方法。比如，在Java 8中，Iterator接口就为remove方法提供了一个默认实现，如下所示：

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException();
    }
}
```

通过这种方式，你可以减少无效的模板代码。实现Iterator接口的每一个类都不需要再声明一个空的remove方法了，因为它现在已经有一个默认的实现。

9.3.2　行为的多继承

默认方法让之前无法想象的事儿以一种优雅的方式得以实现，即行为的多继承。这是一种让类从多个来源重用代码的能力，如图9-3所示。

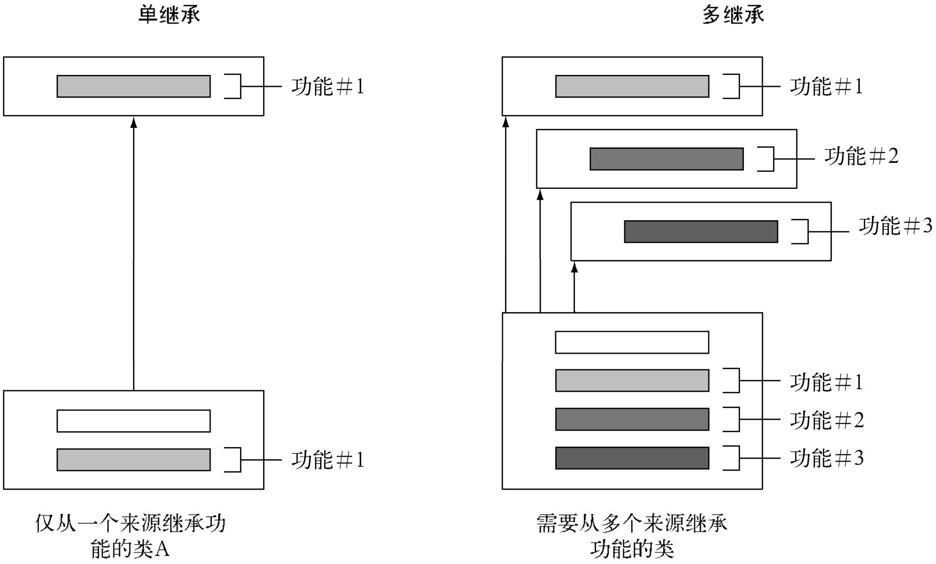


图9-3 单继承和多继承的比较

Java的类只能继承单一的类，但是一个类可以实现多接口。要确认也很简单，下面是Java API中对ArrayList类的定义：

```
public class ArrayList<E> extends AbstractList<E>          ← 继承唯一一个类
    implements List<E>, RandomAccess, Cloneable,
               Serializable, Iterable<E>, Collection<E> {  ← 但是实现了六个接口
}
```

9

1. 类型的多继承

这个例子中ArrayList继承了一个类，实现了六个接口。因此ArrayList实际是七个类型的直接子类，分别是：AbstractList、List、RandomAccess、Cloneable、Serializable、Iterable和Collection。所以，在某种程度上，我们早就有了类型的多继承。

由于Java 8中接口方法可以包含实现，类可以从多个接口中继承它们的行为（即实现的代码）。让我们从一个例子入手，看看如何充分利用这种能力来为我们服务。保持接口的精致性和正交性能帮助你在现有的代码基上最大程度地实现代码复用和行为组合。

2. 利用正交方法的精简接口

假设你需要为你正在创建的游戏定义多个具有不同特质的形状。有的形状需要调整大小，但是不需要有旋转的功能；有的需要能旋转和移动，但是不需要调整大小。这种情况下，你怎么设计才能尽可能地重用代码？

你可以定义一个单独的Rotatable接口，并提供两个抽象方法setRotationAngle和getRotationAngle，如下所示：

```
public interface Rotatable {
    void setRotationAngle(int angleInDegrees);
    int getRotationAngle();
    default void rotateBy(int angleInDegrees){
        setRotationAngle((getRotationAngle () + angle) % 360);
    }
}
```

retateBy **方法的一个默认实现**

这种方式和模板设计模式有些相似，都是以其他方法需要实现的方法定义好框架算法。

现在，实现了Rotatable的所有类都需要提供setRotationAngle和getRotationAngle的实现，但与此同时它们也会天然地继承rotateBy的默认实现。

类似地，你可以定义之前看到的两个接口Moveable和Resizable。它们都包含了默认实现。下面是Moveable的代码：

```
public interface Moveable {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);

    default void moveHorizontally(int distance){
        setX(getX() + distance);
    }

    default void moveVertically(int distance){
        setY(getY() + distance);
    }
}
```

下面是Resizable的代码：

```
public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);

    default void setRelativeSize(int wFactor, int hFactor){
        setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
    }
}
```

3. 组合接口

通过组合这些接口，你现在可以为你的游戏创建不同的实体类。比如，Monster可以移动、旋转和缩放。

```
public class Monster implements Rotatable, Moveable, Resizable {
…
}
```

需要给出所有抽象方法的实现，但无需重复实现默认方法

Monster类会自动继承Rotatable、Moveable和Resizable接口的默认方法。这个例子中，

Monster继承了rotateBy、moveHorizontally、moveVertically和setRelativeSize的实现。

你现在可以直接调用不同的方法：

```
Monster m = new Monster();     ← 构造函数会设置Monster的坐标、高度、宽度及默认仰角
m.rotateBy(180);               ← 调用由Rotatable中继承而来的rotateBy
m.moveVertically(10);          ← 调用由Moveable中继承而来的moveVertically
```

假设你现在需要声明另一个类，它要能移动和旋转，但是不能缩放，比如说Sun。这时也无需复制粘贴代码，你可以像下面这样复用Moveable和Rotatable接口的默认实现。图9-4是这一场景的UML图表。

```
public class Sun implements Moveable, Rotatable {
…                              ← 需要给出所有抽象方法的实现，但无需重复实现默认方法
}
```

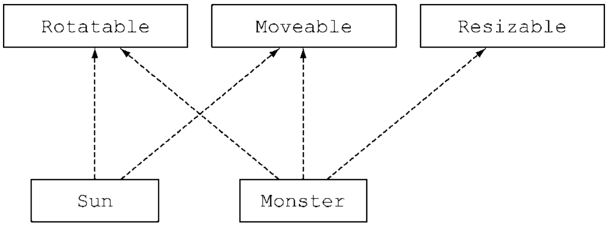


图9-4 多种行为的组合

像你的游戏代码那样使用默认实现来定义简单的接口还有另一个好处。假设你需要修改moveVertically的实现，让它更高效地运行。你可以在Moveable接口内直接修改它的实现，所有实现该接口的类会自动继承新的代码（这里我们假设用户并未定义自己的方法实现）。

关于继承的一些错误观点

继承不应该成为你一谈到代码复用就试图倚靠的万精油。比如，从一个拥有100个方法及字段的类进行继承就不是个好主意，因为这其实会引入不必要的复杂性。你完全可以使用**代理**有效地规避这种窘境，即创建一个方法通过该类的成员变量直接调用该类的方法。这就是为什么有的时候我们发现有些类被刻意地声明为final类型：声明为final的类不能被其他的类继承，避免发生这样的反模式，防止核心代码的功能被污染。注意，有的时候声明为final的类都会有其不同的原因，比如，String类被声明为final，因为我们不希望有人对这样的核心功能产生干扰。

这种思想同样也适用于使用默认方法的接口。通过精简的接口，你能获得最有效的组合，因为你可以只选择你需要的实现。

通过前面的介绍，你已经了解了默认方法多种强大的使用模式。不过也可能还有一些疑惑：如果一个类同时实现了两个接口，这两个接口恰巧又提供了同样的默认方法签名，这时会发生什么情况？类会选择使用哪一个方法？这些问题，我们会在接下来的一节进行讨论。

9.4　解决冲突的规则

我们知道Java语言中一个类只能继承一个父类，但是一个类可以实现多个接口。随着默认方法在Java 8中引入，有可能出现一个类继承了多个方法而它们使用的却是同样的函数签名。这种情况下，类会选择使用哪一个函数？在实际情况中，像这样的冲突可能极少发生，但是一旦发生这样的状况，必须要有一套规则来确定按照什么样的约定处理这些冲突。这一节中，我们会介绍Java编译器如何解决这种潜在的冲突。我们试图回答像“接下来的代码中，哪一个`hello`方法是被`C`类调用的”这样的问题。注意，接下来的例子主要用于说明容易出问题的场景，并不表示这些场景在实际开发过程中会经常发生。

```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}
public class C implements B, A {
    public static void main(String... args) {
        new C().hello();                          <-- 猜猜打印输出的是什么？
    }
}
```

此外，你可能早就对C++语言中著名的菱形继承问题有所了解，菱形继承问题中一个类同时继承了具有相同函数签名的两个方法。到底该选择哪一个实现呢？ Java 8也提供了解决这个问题的方案。请接着阅读下面的内容。

9.4.1　解决问题的三条规则

如果一个类使用相同的函数签名从多个地方（比如另一个类或接口）继承了方法，通过三条规则可以进行判断。

(1) 类中的方法优先级最高。类或父类中声明的方法的优先级高于任何声明为默认方法的优先级。

(2) 如果无法依据第一条进行判断，那么子接口的优先级更高：函数签名相同时，优先选择拥有最具体实现的默认方法的接口，即如果`B`继承了`A`，那么`B`就比`A`更加具体。

(3) 最后，如果还是无法判断，继承了多个接口的类必须通过显式覆盖和调用期望的方法，

显式地选择使用哪一个默认方法的实现。

我们保证，这些就是你需要知道的全部！让我们一起看几个例子。

9.4.2 选择提供了最具体实现的默认方法的接口

让我们回顾一下本节开头的例子，这个例子中C类同时实现了B接口和A接口，而这两个接口恰巧又都定义了名为hello的默认方法。另外，B继承自A。图9-5是这个场景的UML图。

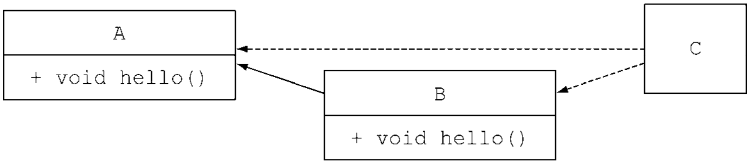


图9-5 提供最具体的默认方法实现的接口，其优先级更高

编译器会使用声明的哪一个hello方法呢？按照规则(2)，应该选择的是提供了最具体实现的默认方法的接口。由于B比A更具体，所以应该选择B的hello方法。所以，程序会打印输出“Hello from B”。

现在，我们看看如果C像下面这样（如图9-6所示）继承自D，会发生什么情况：

```
public class D implements A{ }

public class C extends D implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

猜猜打印输出的是什么？

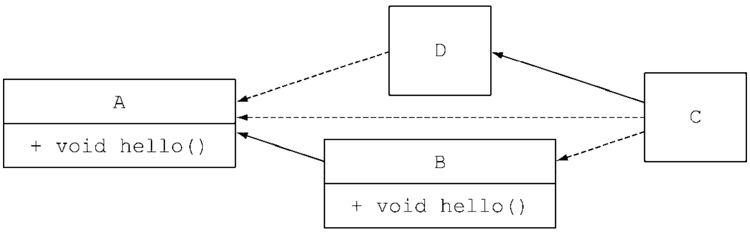


图9-6 继承一个类，实现两个接口的情况

依据规则(1)，类中声明的方法具有更高的优先级。D并未覆盖hello方法，可是它实现了接口A。所以它就拥有了接口A的默认方法。规则(2)说如果类或者父类没有对应的方法，那么就应该选择提供了最具体实现的接口中的方法。因此，编译器会在接口A和接口B的hello方法之间做选择。由于B更加具体，所以程序会再次打印输出“Hello from B”。你可以继续尝试测验9.2，考察一下你对这些规则的理解。

测验9.2：牢记这些判断的规则

我们在这个测验中继续复用之前的例子，唯一的不同在于D现在显式地覆盖了从A接口中继承的hello方法。你认为现在的输出会是什么呢？

```
public class D implements A{
    void hello(){
        System.out.println("Hello from D");
    }
}

public class C extends D implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

答案：由于依据规则(1)，父类中声明的方法具有更高的优先级，所以程序会打印输出“Hello from D”。

注意，D的声明如下：

```
public abstract class D implements A {
    public abstract void hello();
}
```

这样的结果是，虽然在结构上，其他的地方已经声明了默认方法的实现，C还是必须提供自己的hello方法。

9.4.3　冲突及如何显式地消除歧义

到目前为止，你看到的这些例子都能够应用前两条判断规则解决。让我们更进一步，假设B不再继承A（如图9-7所示）：

```
public interface A {
    void hello() {
        System.out.println("Hello from A");
    }
}

public interface B {
    void hello() {
        System.out.println("Hello from B");
    }
}

public class C implements B, A { }
```

这时规则(2)就无法进行判断了，因为从编译器的角度看没有哪一个接口的实现更加具体，两个都差不多。A接口和B接口的hello方法都是有效的选项。所以，Java编译器这时就会抛出一个编译错误，因为它无法判断哪一个方法更合适：“Error: class C inherits unrelated defaults for hello() from types B and A.”

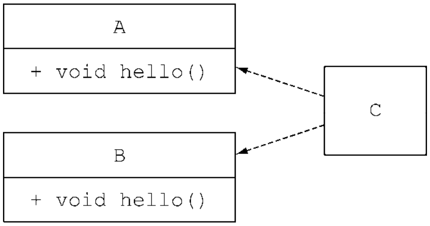


图9-7　同时实现具有相同函数声明的两个接口

冲突的解决

解决这种两个可能的有效方法之间的冲突，没有太多方案；你只能显式地决定你希望在C中使用哪一个方法。为了达到这个目的，你可以覆盖类C中的hello方法，在它的方法体内显式地调用你希望调用的方法。Java 8中引入了一种新的语法X.super.m(…)，其中X是你希望调用的m方法所在的父接口。举例来说，如果你希望C使用来自于B的默认方法，它的调用方式看起来就如下所示：

```
public class C implements B, A {
    void hello(){
        B.super.hello();          <—  显式地选择调用接口B中的方法
    }
}
```

让我们继续看看测验9.3，这是一个相关但更加复杂的例子。

测验9.3：几乎完全一样的函数签名

这个测试中，我们假设接口A和B的声明如下所示：

```
public interface A{
    default Number getNumber(){
        return 10;
    }
}
public interface B{
    default Integer getNumber(){
        return 42;
    }
}
```

类C的声明如下：

```
public class C implements B, A {
    public static void main(String... args) {
        System.out.println(new C().getNumber());
    }
}
```

这个程序的会打印输出什么呢？

答案：类C无法判断A或者B到底哪一个更加具体。这就是类C无法通过编译的原因。

9.4.4 菱形继承问题

让我们考虑最后一种场景，它亦是C++里中最令人头痛的难题。

```
public interface A{
    default void hello(){
        System.out.println("Hello from A");
    }
}

public interface B extends A { }

public interface C extends A { }

public class D implements B, C {
    public static void main(String... args) {
        new D().hello();        // 猜猜打印输出的是什么？
    }
}
```

图9-8以UML图的方式描述了出现这种问题的场景。这种问题叫“菱形问题”，因为类的继承关系图形状像菱形。这种情况下类D中的默认方法到底继承自什么地方 ——源自B的默认方法，还是源自C的默认方法？实际上只有一个方法声明可以选择。只有A声明了一个默认方法。由于这个接口是D的父接口，代码会打印输出“Hello from A”。

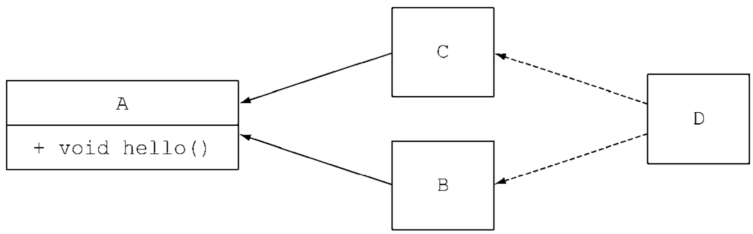


图9-8　菱形问题

现在，我们看看另一种情况，如果B中也提供了一个默认的hello方法，并且函数签名跟A中的方法也完全一致，这时会发生什么情况呢？根据规则(2)，编译器会选择提供了更具体实现的接口中的方法。由于B比A更加具体，所以编译器会选择B中声明的默认方法。如果B和C都使用相同的函数签名声明了hello方法，就会出现冲突，正如我们之前所介绍的，你需要显式地指定使用哪个方法。

顺便提一句，如果你在C接口中添加一个抽象的hello方法（这次添加的不是一个默认方法），会发生什么情况呢？你可能也想知道答案。

```
public interface C extends A {
    void hello();
}
```

这个新添加到C接口中的抽象方法hello比由接口A继承而来的hello方法拥有更高的优先级，

因为C接口更加具体。因此，类D现在需要为hello显式地添加实现，否则该程序无法通过编译。

C++语言中的菱形问题

C++语言中的菱形问题要复杂得多。首先，C++允许类的多继承。默认情况下，如果类D继承了类B和类C，而类B和类C又都继承自类A，类D实际直接访问的是B对象和C对象的副本。最后的结果是，要使用A中的方法必须显式地声明：这些方法来自于B接口，还是来自于C接口。此外，类也有状态，所以修改B的成员变量不会在C对象的副本中反映出来。

现在你应该已经了解了，如果一个类的默认方法使用相同的函数签名继承自多个接口，解决冲突的机制其实相当简单。你只需要遵守下面这三条准则就能解决所有可能的冲突。

- 首先，类或父类中显式声明的方法，其优先级高于所有的默认方法。
- 如果用第一条无法判断，方法签名又没有区别，那么选择提供最具体实现的默认方法的接口。
- 最后，如果冲突依旧无法解决，你就只能在你的类中覆盖该默认方法，显式地指定在你的类中使用哪一个接口中的方法。

9.5 小结

下面是本章你应该掌握的关键概念。

- Java 8中的接口可以通过默认方法和静态方法提供方法的代码实现。
- 默认方法的开头以关键字default修饰，方法体与常规的类方法相同。
- 向发布的接口添加抽象方法不是源码兼容的。
- 默认方法的出现能帮助库的设计者以后向兼容的方式演进API。
- 默认方法可以用于创建可选方法和行为的多继承。
- 我们有办法解决由于一个类从多个接口中继承了拥有相同函数签名的方法而导致的冲突。
- 类或者父类中声明的方法的优先级高于任何默认方法。如果前一条无法解决冲突，那就选择同函数签名的方法中实现得最具体的那个接口的方法。
- 两个默认方法都同样具体时，你需要在类中覆盖该方法，显式地选择使用哪个接口中提供的默认方法。

第 10 章

用Optional取代null

本章内容

- null引用引发的问题，以及为什么要避免null引用
- 从null到Optional：以null安全的方式重写你的域模型
- 让Optional发光发热： 去除代码中对null的检查
- 读取Optional中可能值的几种方法
- 对可能缺失值的再思考

如果你作为Java程序员曾经遭遇过NullPointerException，请举起手。如果这是你最常遭遇的异常，请继续举手。非常可惜，这个时刻，我们无法看到对方，但是我相信很多人的手这个时刻是举着的。我们还猜想你可能也有这样的想法："毫无疑问，我承认，对任何一位Java程序员来说，无论是初出茅庐的新人，还是久经江湖的专家，NullPointerException都是他心中的痛，可是我们又无能为力，因为这就是我们为了使用方便甚至不可避免的像null引用这样的构造所付出的代价。"这就是程序设计世界里大家都持有的观点，然而，这可能并非事实的全部真相，只是我们根深蒂固的一种偏见。

1965年，英国一位名为Tony Hoare的计算机科学家在设计ALGOL W语言时提出了null引用的想法。ALGOL W是第一批在堆上分配记录的类型语言之一。Hoare选择null引用这种方式，"只是因为这种方法实现起来非常容易"。虽然他的设计初衷就是要"通过编译器的自动检测机制，确保所有使用引用的地方都是绝对安全的"，他还是决定为null引用开个绿灯，因为他认为这是为"不存在的值"建模最容易的方式。很多年后，他开始为自己曾经做过这样的决定而后悔不迭，把它称为"我价值百万的重大失误"。我们已经看到它带来的后果——程序员对对象的字段进行检查，判断它的值是否为期望的格式，最终却发现我们查看的并不是一个对象，而是一个空指针，它会立即抛出一个让人厌烦的NullPointerException异常。

实际上，Hoare的这段话低估了过去五十年来数百万程序员为修复空引用所耗费的代价。近十年出现的大多数现代程序设计语言[①]，包括Java，都采用了同样的设计方式，其原因是为了与

① 为数不多的几个最著名的例外是典型的函数式语言，比如Haskell、ML；这些语言中引入了代数数据类型，允许显式地声明数据类型，明确地定义了特殊变量值（比如null）能否使用在定义类型的类型（type-by-type basis）中。

更老的语言保持兼容，或者就像Hoare曾经陈述的那样，“仅仅是因为这样实现起来更加容易”。让我们从一个简单的例子入手，看看使用null都有什么样的问题。

10.1　如何为缺失的值建模

假设你需要处理下面这样的嵌套对象，这是一个拥有汽车及汽车保险的客户。

代码清单10-1　Person/Car/Insurance的数据模型

```
public class Person {
    private Car car;
    public Car getCar() { return car; }
}

public class Car {
    private Insurance insurance;
    public Insurance getInsurance() { return insurance; }
}

public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

那么，下面这段代码存在怎样的问题呢？

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

这段代码看起来相当正常，但是现实生活中很多人没有车。所以调用getCar方法的结果会怎样呢？在实践中，一种比较常见的做法是返回一个null引用，表示该值的缺失，即用户没有车。而接下来，对getInsurance的调用会返回null引用的insurance，这会导致运行时出现一个NullPointerException，终止程序的运行。但这还不是全部。如果返回的person值为null会怎样？如果getInsurance的返回值也是null，结果又会怎样？

10.1.1　采用防御式检查减少 NullPointerException

怎样做才能避免这种不期而至的NullPointerException呢？通常，你可以在需要的地方添加null的检查（过于激进的防御式检查甚至会在不太需要的地方添加检测代码），并且添加的方式往往各有不同。下面这个例子是我们试图在方法中避免NullPointerException的第一次尝试。

代码清单10-2　null-安全的第一种尝试：深层质疑

```
public String getCarInsuranceName(Person person) {
    if (person != null) {                              <-
        Car car = person.getCar();                        每个null检查都会增加调
        if (car != null) {                             <- 用链上剩余代码的嵌套层数
            Insurance insurance = car.getInsurance();
            if (insurance != null) {                   <-
                return insurance.getName();
```

```
            }
        }
    }
    return "Unknown";
}
```

这个方法每次引用一个变量都会做一次null检查，如果引用链上的任何一个遍历的解变量值为null，它就返回一个值为“Unknown”的字符串。唯一的例外是保险公司的名字，你不需要对它进行检查，原因很简单，因为任何一家公司必定有个名字。注意到了吗，由于你掌握业务领域的知识，避免了最后这个检查，但这并不会直接反映在你建模数据的Java类之中。

我们将代码清单10-2标记为“深层质疑”，原因是它不断重复着一种模式：每次你不确定一个变量是否为null时，都需要添加一个进一步嵌套的if块，也增加了代码缩进的层数。很明显，这种方式不具备扩展性，同时还牺牲了代码的可读性。面对这种窘境，你也许愿意尝试另一种方案。下面的代码清单中，我们试图通过一种不同的方式避免这种问题。

代码清单10-3　null-安全的第二种尝试：过多的退出语句

```
public String getCarInsuranceName(Person person) {
    if (person == null) {                          <-- 每个null检查都
        return "Unknown";                              会添加新的退出点
    }
    Car car = person.getCar();
    if (car == null) {                             <--
        return "Unknown";                             | 每个null检查都
    }                                                 | 会添加新的退出点
    Insurance insurance = car.getInsurance();
    if (insurance == null) {                       <--
        return "Unknown";
    }
    return insurance.getName();
}
```

第二种尝试中，你试图避免深层递归的if语句块，采用了一种不同的策略：每次你遭遇null变量，都返回一个字符串常量“Unknown”。然而，这种方案远非理想，现在这个方法有了四个截然不同的退出点，使得代码的维护异常艰难。更糟的是，发生null时返回的默认值，即字符串“Unknown”在三个不同的地方重复出现——出现拼写错误的概率不小！当然，你可能会说，我们可以用把它们抽取到一个常量中的方式避免这种问题。

进一步而言，这种流程是极易出错的；如果你忘记检查了那个可能为null的属性会怎样？通过这一章的学习，你会了解使用null来表示变量值的缺失是大错特错的。你需要更优雅的方式来对缺失的变量值建模。

10.1.2　null 带来的种种问题

让我们一起回顾一下到目前为止进行的讨论，在Java程序开发中使用null会带来理论和实际操作上的种种问题。

- 它是错误之源。
  NullPointerException是目前Java程序开发中最典型的异常。
- 它会使你的代码膨胀。
  它让你的代码充斥着深度嵌套的null检查，代码的可读性糟糕透顶。
- 它自身是毫无意义的。
  null自身没有任何的语义，尤其是，它代表的是在静态类型语言中以一种错误的方式对缺失变量值的建模。
- 它破坏了Java的哲学。
  Java一直试图避免让程序员意识到指针的存在，唯一的例外是：null指针。
- 它在Java的类型系统上开了个口子。
  null并不属于任何类型，这意味着它可以被赋值给任意引用类型的变量。这会导致问题，原因是当这个变量被传递到系统中的另一个部分后，你将无法获知这个null变量最初的赋值到底是什么类型。

为了解业界针对这个问题给出的解决方案，我们一起简单看看其他语言提供了哪些功能。

10.1.3　其他语言中 null 的替代品

近年来出现的语言，比如Groovy，通过引入安全导航操作符（Safe Navigation Operator，标记为?)可以安全访问可能为null的变量。为了理解它是如何工作的，让我们看看下面这段Groovy代码，它的功能是获取某个用户替他的车保险的保险公司的名称：

```
def carInsuranceName = person?.car?.insurance?.name
```


这段代码的表述相当清晰。person对象可能没有car对象，你试图通过赋一个null给Person对象的car引用，对这种可能性建模。类似地，car也可能没有insurance。Groovy的安全导航操作符能够避免在访问这些可能为null引用的变量时抛出NullPointerException，在调用链中的变量遭遇null时将null引用沿着调用链传递下去，返回一个null。

关于Java 7的讨论中曾经建议过一个类似的功能，不过后来又被舍弃了。不知道为什么，我们在Java中似乎并不特别期待出现一种安全导航操作符，几乎所有的Java程序员碰到NullPointerException时的第一冲动就是添加一个if语句，在调用方法使用该变量之前检查它的值是否为null，快速地搞定问题。如果你按照这种方式解决问题，丝毫不考虑你的算法或者你的数据模型在这种状况下是否应该返回一个null，那么你其实并没有真正解决这个问题，只是暂时地掩盖了问题，使得下次该问题的调查和修复更加困难，而你很可能就是下个星期或下个月要面对这个问题的人。刚才的那种方式实际上是掩耳盗铃，只是在清扫地毯下的灰尘。而Groovy的null安全解引用操作符也只是一个更强大的扫把，让我们可以毫无顾忌地犯错。你不会忘记做这样的检查，因为类型系统会强制你进行这样的操作。

另一些函数式语言，比如Haskell、Scala，试图从另一个角度处理这个问题。Haskell中包含了一个Maybe类型，它本质上是对optional值的封装。Maybe类型的变量可以是指定类型的值，

也可以什么都不是。但是它并没有null引用的概念。Scala有类似的数据结构,名字叫Option[T],它既可以包含类型为T的变量，也可以不包含该变量，我们在第15章会详细讨论这种类型。要使用这种类型，你必须显式地调用Option类型的available操作，检查该变量是否有值，而这其实也是一种变相的“null检查”。

好了，我们似乎有些跑题了，刚才这些听起来都十分抽象。你可能会疑惑:“那么Java 8提供了什么呢？”嗯，实际上Java 8从“optional值”的想法中吸取了灵感，引入了一个名为java.util.Optional<T>的新的类。这一章里,我们会展示使用这种方式对可能缺失的值建模,而不是直接将null赋值给变量所带来的好处。我们还会阐释从null到Optional的迁移,你需要反思的是:如何在你的域模型中使用optional值。最后,我们会介绍新的Optional类提供的功能,并附几个实际的例子,展示如何有效地使用这些特性。最终,你会学会如何设计更好的API——用户只需要阅读方法签名就能知道它是否接受一个optional的值。

10.2　Optional 类入门

汲取Haskell和Scala的灵感，Java 8中引入了一个新的类java.util.Optional<T>。这是一个封装Optional值的类。举例来说，使用新的类意味着，如果你知道一个人可能有也可能没有车，那么Person类内部的car变量就不应该声明为Car，遭遇某人没有车时把null引用赋值给它，而是应该像图10-1那样直接将其声明为Optional<Car>类型。

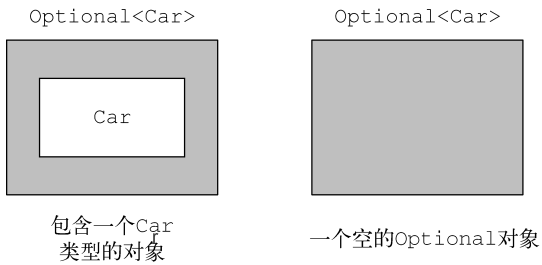


图10-1　使用Optional定义的Car类

变量存在时，Optional类只是对类简单封装。变量不存在时，缺失的值会被建模成一个“空”的Optional对象，由方法Optional.empty()返回。Optional.empty()方法是一个静态工厂方法，它返回Optional类的特定单一实例。你可能还有疑惑，null引用和Optional.empty()有什么本质的区别吗？从语义上，你可以把它们当作一回事儿，但是实际中它们之间的差别非常大：如果你尝试解引用一个null，一定会触发NullPointerException，不过使用Optional.empty()就完全没事儿，它是Optional类的一个有效对象，多种场景都能调用，非常有用。关于这一点，接下来的部分会详细介绍。

使用Optional而不是null的一个非常重要而又实际的语义区别是，第一个例子中，我们在声明变量时使用的是Optional<Car>类型，而不是Car类型，这句声明非常清楚地表明了这

里发生变量缺失是允许的。与此相反，使用Car这样的类型，可能将变量赋值为null，这意味着你需要独立面对这些，你只能依赖你对业务模型的理解，判断一个null是否属于该变量的有效范畴。

牢记上面这些原则，你现在可以使用Optional类对代码清单10-1中最初的代码进行重构，结果如下。

代码清单10-4 使用Optional重新定义Person/Car/Insurance的数据模型

```
public class Person {
    private Optional<Car> car;                          <-- 人可能有车，也可能没有车，因此将这个字段声明为Optional
    public Optional<Car> getCar() { return car; }
}

public class Car {
    private Optional<Insurance> insurance;              <-- 车可能进行了保险，也可能没有保险，所以将这个字段声明为Optional
    public Optional<Insurance> getInsurance() { return insurance; }
}

public class Insurance {
    private String name;                                <-- 保险公司必须有名字
    public String getName() { return name; }
}
```

发现Optional是如何丰富你模型的语义了吧。代码中person引用的是Optional<Car>，而car引用的是Optional<Insurance>，这种方式非常清晰地表达了你的模型中一个person可能拥有也可能没有car的情形，同样，car可能进行了保险，也可能没有保险。

与此同时，我们看到insurance公司的名称被声明成String类型，而不是Optional-<String>，这非常清楚地表明声明为insurance公司的类型必须提供公司名称。使用这种方式，一旦解引用insurance公司名称时发生NullPointerException，你就能非常确定地知道出错的原因，不再需要为其添加null的检查，因为null的检查只会掩盖问题，并未真正地修复问题。insurance公司必须有个名字，所以，如果你遇到一个公司没有名称，你需要调查你的数据出了什么问题，而不应该再添加一段代码，将这个问题隐藏。

在你的代码中始终如一地使用Optional，能非常清晰地界定出变量值的缺失是结构上的问题，还是你算法上的缺陷，抑或是你数据中的问题。另外，我们还想特别强调，引入Optional类的意图并非要消除每一个null引用。与此相反，它的目标是帮助你更好地设计出普适的API，让程序员看到方法签名，就能了解它是否接受一个Optional的值。这种强制会让你更积极地将变量从Optional中解包出来，直面缺失的变量值。

10.3 应用 Optional 的几种模式

到目前为止，一切都很顺利；你已经知道了如何使用Optional类型来声明你的域模型，也了解了这种方式与直接使用null引用表示变量值的缺失的优劣。但是，我们该如何使用呢？用这种方式能做什么，或者怎样使用Optional封装的值呢？

10.3.1　创建 Optional 对象

使用Optional之前，你首先需要学习的是如何创建Optional对象。完成这一任务有多种方法。

1. 声明一个空的Optional

正如前文已经提到，你可以通过静态工厂方法Optional.empty，创建一个空的Optional对象：

```
Optional<Car> optCar = Optional.empty();
```

2. 依据一个非空值创建Optional

你还可以使用静态工厂方法Optional.of，依据一个非空值创建一个Optional对象：

```
Optional<Car> optCar = Optional.of(car);
```

如果car是一个null，这段代码会立即抛出一个NullPointerException，而不是等到你试图访问car的属性值时才返回一个错误。

3. 可接受null的Optional

最后，使用静态工厂方法Optional.ofNullable，你可以创建一个允许null值的Optional对象：

```
Optional<Car> optCar = Optional.ofNullable(car);
```

如果car是null，那么得到的Optional对象就是个空对象。

你可能已经猜到，我们还需要继续研究"如何获取Optional变量中的值"。尤其是，Optional提供了一个get方法，它能非常精准地完成这项工作，我们在后面会详细介绍这部分内容。不过get方法在遭遇到空的Optional对象时也会抛出异常，所以不按照约定的方式使用它，又会让我们再度陷入由null引起的代码维护的梦魇。因此，我们首先从无需显式检查的Optional值的使用入手，这些方法与Stream中的某些操作极其相似。

10.3.2　使用 map 从 Optional 对象中提取和转换值

从对象中提取信息是一种比较常见的模式。比如，你可能想要从insurance公司对象中提取公司的名称。提取名称之前，你需要检查insurance对象是否为null，代码如下所示：

```
String name = null;
if(insurance != null){
    name = insurance.getName();
}
```

为了支持这种模式，Optional提供了一个map方法。它的工作方式如下（这里，我们继续借用了代码清单10-4的模式）：

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```

从概念上，这与我们在第4章和第5章中看到的流的map方法相差无几。map操作会将提供的函数应用于流的每个元素。你可以把Optional对象看成一种特殊的集合数据，它至多包含一个元素。如果Optional包含一个值，那函数就将该值作为参数传递给map，对该值进行转换。如果Optional为空，就什么也不做。图10-2对这种相似性进行了说明，展示了把一个将正方形转换为三角形的函数，分别传递给正方形和Optional正方形流的map方法之后的结果。

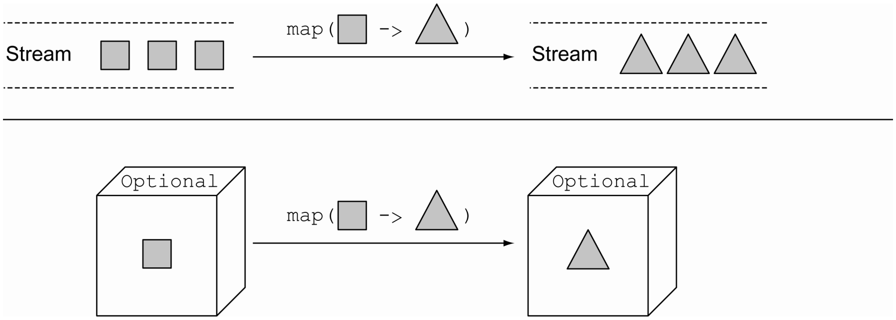


图10-2 Stream和Optional的map方法对比

这看起来挺有用，但是你怎样才能应用起来，重构之前的代码呢？前文的代码里用安全的方式链接了多个方法。

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

为了达到这个目的，我们需要求助Optional提供的另一个方法flatMap。

10.3.3 使用 flatMap 链接 Optional 对象

由于我们刚刚学习了如何使用map，你的第一反应可能是我们可以利用map重写之前的代码，如下所示：

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name =
    optPerson.map(Person::getCar)
             .map(Car::getInsurance)
             .map(Insurance::getName);
```

不幸的是，这段代码无法通过编译。为什么呢？optPerson是Optional<Person>类型的变量，调用map方法应该没有问题。但getCar返回的是一个Optional<Car>类型的对象（如代码清单10-4所示），这意味着map操作的结果是一个Optional<Optional<Car>>类型的对象。因此，它对getInsurance的调用是非法的，因为最外层的optional对象包含了另一个optional对象的值，而它当然不会支持getInsurance方法。图10-3说明了你会遭遇的嵌套式optional结构。

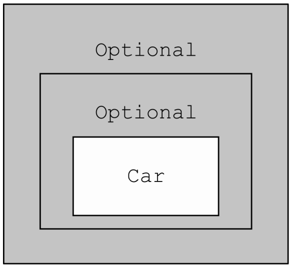


图10-3　两层的optional对象

所以，我们该如何解决这个问题呢？让我们再回顾一下你刚刚在流上使用过的模式：flatMap方法。使用流时，flatMap方法接受一个函数作为参数，这个函数的返回值是另一个流。这个方法会应用到流中的每一个元素，最终形成一个新的流的流。但是flagMap会用流的内容替换每个新生成的流。换句话说，由方法生成的各个流会被合并或者扁平化为一个单一的流。这里你希望的结果其实也是类似的，但是你想要的是将两层的optional合并为一个。

跟图10-2类似，我们借助图10-4来说明flatMap方法在Stream和Optional类之间的相似性。

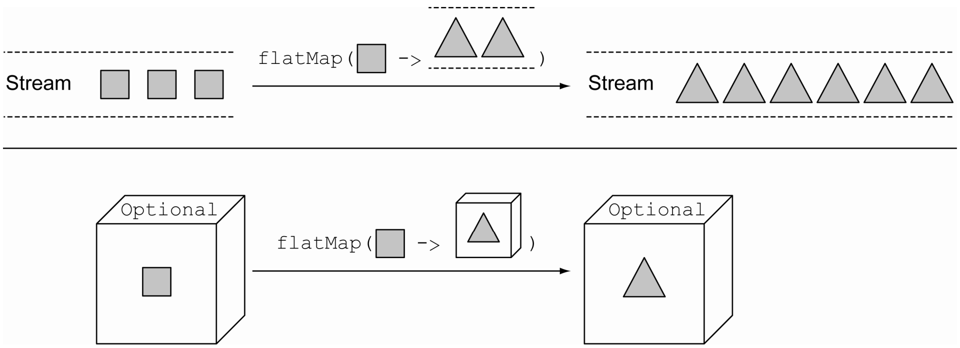


图10-4　Stream和Optional的flagMap方法对比

这个例子中，传递给流的flatMap方法会将每个正方形转换为另一个流中的两个三角形。那么，map操作的结果就包含有三个新的流，每一个流包含两个三角形，但flatMap方法会将这种两层的流合并为一个包含六个三角形的单一流。类似地，传递给optional的flatMap方法的函数会将原始包含正方形的optional对象转换为包含三角形的optional对象。如果将该方法传递给map方法，结果会是一个Optional对象，而这个Optional对象中包含了三角形；但flatMap方法会将这种两层的Optional对象转换为包含三角形的单一Optional对象。

1. 使用Optional获取car的保险公司名称

相信现在你已经对Optional的map和flatMap方法有了一定的了解，让我们看看如何应用。代码清单10-2和代码清单10-3的示例用基于Optional的数据模式重写之后，如代码清单10-5所示。

代码清单10-5　使用Optional获取car的Insurance名称

```
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");          <-- 如果Optional的结果值为空，设置默认值
}
```

通过比较代码清单10-5和之前的两个代码清单，我们可以看到，处理潜在可能缺失的值时，使用Optional具有明显的优势。这一次，你可以用非常容易却又普适的方法实现之前你期望的效果——不再需要使用那么多的条件分支，也不会增加代码的复杂性。

从具体的代码实现来看，首先我们注意到你修改了代码清单10-2和代码清单10-3中的getCarInsuranceName方法的签名，因为我们很明确地知道存在这样的用例，即一个不存在的Person被传递给了方法，比如，Person是使用某个标识符从数据库中查询出来的，你想要对数据库中不存在指定标识符对应的用户数据的情况进行建模。你可以将方法的参数类型由Person改为Optional<Person>，对这种特殊情况进行建模。

我们再一次看到这种方式的优点，它通过类型系统让你的域模型中隐藏的知识显式地体现在你的代码中，换句话说，你永远都不应该忘记语言的首要功能就是沟通，即使对程序设计语言而言也没有什么不同。声明方法接受一个Optional参数，或者将结果作为Optional类型返回，让你的同事或者未来你方法的使用者，很清楚地知道它可以接受空值，或者它可能返回一个空值。

2. 使用Optional解引用串接的Person/Car/Insurance对象

由Optional<Person>对象，我们可以结合使用之前介绍的map和flatMap方法，从Person中解引用出Car，从Car中解引用出Insurance，从Insurance对象中解引用出包含insurance公司名称的字符串。图10-5对这种流水线式的操作进行了说明。

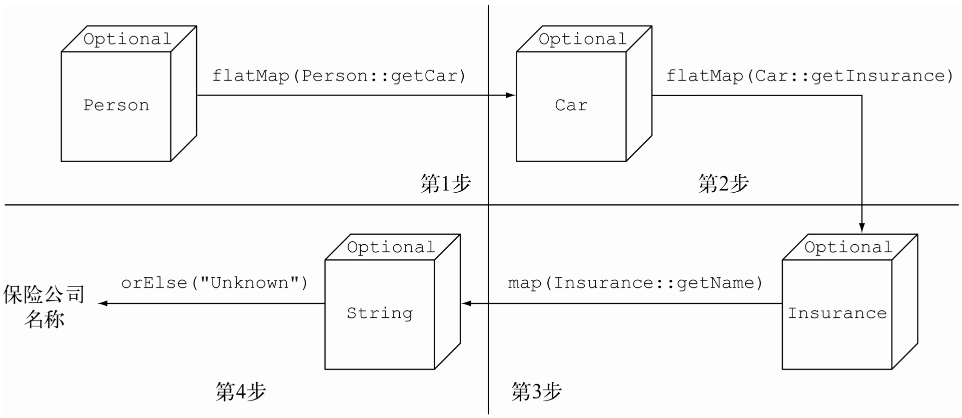


图10-5　使用Optional解引用串接的Person/Car/Insurance

这里，我们从以Optional封装的Person入手，对其调用flatMap(Person::getCar)。如前所述，这种调用逻辑上可以划分为两步。第一步，某个Function作为参数，被传递给由

Optional封装的Person对象，对其进行转换。这个场景中，Function的具体表现是一个方法引用，即对Person对象的getCar方法进行调用。由于该方法返回一个Optional<Car>类型的对象，Optional内的Person也被转换成了这种对象的实例，结果就是一个两层的Optional对象，最终它们会被flagMap操作合并。从纯理论的角度而言，你可以将这种合并操作简单地看成把两个Optional对象结合在一起，如果其中有一个对象为空，就构成一个空的Optional对象。如果你对一个空的Optional对象调用flatMap,实际情况又会如何呢？结果不会发生任何改变，返回值也是个空的Optional对象。与此相反，如果Optional封装了一个Person对象，传递给flapMap的Function，就会应用到Person上对其进行处理。这个例子中，由于Function的返回值已经是一个Optional对象，flapMap方法就直接将其返回。

第二步与第一步大同小异，它会将Optional<Car>转换为Optional<Insurance>。第三步则会将Optional<Insurance>转化为Optional<String>对象，由于Insurance.getName()方法的返回类型为String，这里就不再需要进行flapMap操作了。

截至目前为止，返回的Optional可能是两种情况：如果调用链上的任何一个方法返回一个空的Optional，那么结果就为空，否则返回的值就是你期望的保险公司的名称。那么，你如何读出这个值呢？毕竟你最后得到的这个对象还是个Optional<String>,它可能包含保险公司的名称，也可能为空。代码清单10-5中，我们使用了一个名为orElse的方法，当Optional的值为空时，它会为其设定一个默认值。除此之外，还有很多其他的方法可以为Optional设定默认值，或者解析出Optional代表的值。接下来我们会对此做进一步的探讨。

在域模型中使用Optional，以及为什么它们无法序列化

在代码清单10-4中，我们展示了如何在你的域模型中使用Optional，将允许缺失或者暂无定义的变量值用特殊的形式标记出来。然而，Optional类设计者的初衷并非如此，他们构思时怀揣的是另一个用例。这一点，Java语言的架构师Brian Goetz曾经非常明确地陈述过，Optional的设计初衷仅仅是要支持能返回Optional对象的语法。

由于Optional类设计时就没特别考虑将其作为类的字段使用，所以它也并未实现Serializable接口。由于这个原因，如果你的应用使用了某些要求序列化的库或者框架，在域模型中使用Optional，有可能引发应用程序故障。然而，我们相信，通过前面的介绍，你已经看到用Optional声明域模型中的某些类型是个不错的主意，尤其是你需要遍历有可能全部或部分为空，或者可能不存在的对象时。如果你一定要实现序列化的域模型，作为替代方案，我们建议你像下面这个例子那样，提供一个能访问声明为Optional、变量值可能缺失的接口，代码清单如下：

```
public class Person {
    private Car car;
    public Optional<Car> getCarAsOptional() {
        return Optional.ofNullable(car);
    }
}
```

10.3.4 默认行为及解引用 Optional 对象

我们决定采用orElse方法读取这个变量的值，使用这种方式你还可以定义一个默认值，遭遇空的Optional变量时，默认值会作为该方法的调用返回值。Optional类提供了多种方法读取Optional实例中的变量值。

- get()是这些方法中最简单但又最不安全的方法。如果变量存在，它直接返回封装的变量值，否则就抛出一个NoSuchElementException异常。所以，除非你非常确定Optional变量一定包含值，否则使用这个方法是个相当糟糕的主意。此外，这种方式即便相对于嵌套式的null检查，也并未体现出多大的改进。
- orElse(T other)是我们在代码清单10-5中使用的方法，正如之前提到的，它允许你在Optional对象不包含值时提供一个默认值。
- orElseGet(Supplier<? extends T> other)是orElse方法的延迟调用版，Supplier方法只有在Optional对象不含值时才执行调用。如果创建默认值是件耗时费力的工作，你应该考虑采用这种方式（借此提升程序的性能），或者你需要非常确定某个方法仅在Optional为空时才进行调用，也可以考虑该方式（这种情况有严格的限制条件）。
- orElseThrow(Supplier<? extends X> exceptionSupplier)和get方法非常类似，它们遭遇Optional对象为空时都会抛出一个异常，但是使用orElseThrow你可以定制希望抛出的异常类型。
- ifPresent(Consumer<? super T>)让你能在变量值存在时执行一个作为参数传入的方法，否则就不进行任何操作。

Optional类和Stream接口的相似之处，远不止map和flatMap这两个方法。还有第三个方法filter，它的行为在两种类型之间也极其相似，我们会在10.3.6节做进一步的介绍。

10

10.3.5 两个 Optional 对象的组合

现在，我们假设你有这样一个方法，它接受一个Person和一个Car对象，并以此为条件对外部提供的服务进行查询，通过一些复杂的业务逻辑，试图找到满足该组合的最便宜的保险公司：

```
public Insurance findCheapestInsurance(Person person, Car car) {
    // 不同的保险公司提供的查询服务
    // 对比所有数据
    return cheapestCompany;
}
```

我们还假设你想要该方法的一个null-安全的版本，它接受两个Optional对象作为参数，返回值是一个Optional<Insurance>对象，如果传入的任何一个参数值为空，它的返回值亦为空。Optional类还提供了一个isPresent方法，如果Optional对象包含值，该方法就返回true，所以你的第一想法可能是通过下面这种方式实现该方法：

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
                              Optional<Person> person, Optional<Car> car) {
    if (person.isPresent() && car.isPresent()) {
```

```
            return Optional.of(findCheapestInsurance(person.get(), car.get()));
        } else {
            return Optional.empty();
        }
    }
```

这个方法具有明显的优势，我们从它的签名就能非常清楚地知道无论是person还是car，它的值都有可能为空，出现这种情况时，方法的返回值也不会包含任何值。不幸的是，该方法的具体实现和你之前曾经实现的null检查太相似了：方法接受一个Person和一个Car对象作为参数，而二者都有可能为null。利用Optional类提供的特性，有没有更好或更地道的方式来实现这个方法呢? 花几分钟时间思考一下测验10.1，试试能不能找到更优雅的解决方案。

测验10.1：以不解包的方式组合两个Optional对象

结合本节中介绍的map和flatMap方法，用一行语句重新实现之前出现的nullSafeFindCheapestInsurance()方法。

答案：你可以像使用三元操作符那样，无需任何条件判断的结构，以一行语句实现该方法，代码如下。

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
                              Optional<Person> person, Optional<Car> car) {
    return person.flatMap(p -> car.map(c -> findCheapestInsurance(p, c)));
}
```

这段代码中，你对第一个Optional对象调用flatMap方法，如果它是个空值，传递给它的Lambda表达式不会执行，这次调用会直接返回一个空的Optional对象。反之，如果person对象存在，这次调用就会将其作为函数Function的输入，并按照与flatMap方法的约定返回一个Optional<Insurance>对象。这个函数的函数体会对第二个Optional对象执行map操作，如果第二个对象不包含car，函数Function就返回一个空的Optional对象，整个nullSafeFindCheapestInsuranc方法的返回值也是一个空的Optional对象。最后，如果person和car对象都存在，作为参数传递给map方法的Lambda表达式能够使用这两个值安全地调用原始的findCheapestInsurance方法，完成期望的操作。

Optional类和Stream接口的相似之处远不止map和flatMap这两个方法。还有第三个方法filter，它的行为在两种类型之间也极其相似，我们在接下来的一节会进行介绍。

10.3.6　使用 filter 剔除特定的值

你经常需要调用某个对象的方法，查看它的某些属性。比如，你可能需要检查保险公司的名称是否为“Cambridge-Insurance”。为了以一种安全的方式进行这些操作，你首先需要确定引用指向的Insurance对象是否为null，之后再调用它的getName方法，如下所示：

```
Insurance insurance = ...;
if(insurance != null && "CambridgeInsurance".equals(insurance.getName())){
   System.out.println("ok");
```

```
}
```

使用Optional对象的filter方法，这段代码可以重构如下：

```
Optional<Insurance> optInsurance = ...;
optInsurance.filter(insurance ->
                        "CambridgeInsurance".equals(insurance.getName()))
            .ifPresent(x -> System.out.println("ok"));
```

filter方法接受一个谓词作为参数。如果Optional对象的值存在，并且它符合谓词的条件，filter方法就返回其值；否则它就返回一个空的Optional对象。如果你还记得我们可以将Optional看成最多包含一个元素的Stream对象，这个方法的行为就非常清晰了。如果Optional对象为空，它不做任何操作，反之，它就对Optional对象中包含的值施加谓词操作。如果该操作的结果为true，它不做任何改变，直接返回该Optional对象，否则就将该值过滤掉，将Optional的值置空。通过测验10.2，可以测试你对filter方法工作方式的理解。

测验10.2：对Optional对象进行过滤

假设在我们的Person/Car/Insurance 模型中，Person还提供了一个方法可以取得Person对象的年龄，请使用下面的签名改写代码清单10-5中的getCarInsuranceName方法：

```
public String getCarInsuranceName(Optional<Person> person, int minAge)
```

找出年龄大于或者等于minAge参数的Person所对应的保险公司列表。

答案：你可以对Optional封装的Person对象进行filter操作，设置相应的条件谓词，即如果person的年龄大于minAge参数的设定值，就返回该值，并将谓词传递给filter方法，代码如下所示。

```
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

下一节中，我们会探讨Optional类剩下的一些特性，并提供更实际的例子，展示多种你能够应用于代码中更好地管理缺失值的技巧。

表10-1对Optional类中的方法进行了分类和概括。

表10-1 Optional类的方法

方　法	描　述
empty	返回一个空的 Optional 实例
filter	如果值存在并且满足提供的谓词，就返回包含该值的 Optional 对象；否则返回一个空的 Optional 对象

（续）

方 法	描 述
flatMap	如果值存在，就对该值执行提供的 mapping 函数调用，返回一个 Optional 类型的值，否则就返回一个空的 Optional 对象
get	如果该值存在，将该值用 Optional 封装返回，否则抛出一个 NoSuchElementException 异常
ifPresent	如果值存在，就执行使用该值的方法调用，否则什么也不做
isPresent	如果值存在就返回 true，否则返回 false
map	如果值存在，就对该值执行提供的 mapping 函数调用
of	将指定值用 Optional 封装之后返回，如果该值为 null，则抛出一个 NullPointerException 异常
ofNullable	将指定值用 Optional 封装之后返回，如果该值为 null，则返回一个空的 Optional 对象
orElse	如果有值则将其返回，否则返回一个默认值
orElseGet	如果有值则将其返回，否则返回一个由指定的 Supplier 接口生成的值
orElseThrow	如果有值则将其返回，否则抛出一个由指定的 Supplier 接口生成的异常

10.4 使用 Optional 的实战示例

相信你已经了解，有效地使用Optional类意味着你需要对如何处理潜在缺失值进行全面的反思。这种反思不仅仅限于你曾经写过的代码，更重要的可能是，你如何与原生Java API实现共存共赢。

实际上，我们相信如果Optional类能够在这些API创建之初就存在的话，很多API的设计编写可能会大有不同。为了保持后向兼容性，我们很难对老的Java API进行改动，让它们也使用Optional，但这并不表示我们什么也做不了。你可以在自己的代码中添加一些工具方法，修复或者绕过这些问题，让你的代码能享受Optional带来的威力。我们会通过几个实际的例子讲解如何达到这样的目的。

10.4.1 用 Optional 封装可能为 null 的值

现存Java API几乎都是通过返回一个null的方式来表示需要值的缺失，或者由于某些原因计算无法得到该值。比如，如果Map中不含指定的键对应的值，它的get方法会返回一个null。但是，正如我们之前介绍的，大多数情况下，你可能希望这些方法能返回一个Optional对象。你无法修改这些方法的签名，但是你很容易用Optional对这些方法的返回值进行封装。我们接着用Map做例子，假设你有一个Map<String, Object>方法，访问由key索引的值时，如果map中没有与key关联的值，该次调用就会返回一个null。

```
Object value = map.get("key");
```

使用Optional封装map的返回值，你可以对这段代码进行优化。要达到这个目的有两种方式：你可以使用笨拙的if-then-else判断语句，毫无疑问这种方式会增加代码的复杂度；或者

你可以采用我们前文介绍的Optional.ofNullable方法：

```
Optional<Object> value = Optional.ofNullable(map.get("key"));
```

每次你希望安全地对潜在为null的对象进行转换，将其替换为Optional对象时，都可以考虑使用这种方法。

10.4.2 异常与 Optional 的对比

由于某种原因，函数无法返回某个值，这时除了返回null，Java API比较常见的替代做法是抛出一个异常。这种情况比较典型的例子是使用静态方法Integer.parseInt(String)，将String转换为int。在这个例子中，如果String无法解析到对应的整型，该方法就抛出一个NumberFormatException。最后的效果是，发生String无法转换为int时，代码发出一个遭遇非法参数的信号，唯一的不同是，这次你需要使用try/catch 语句，而不是使用if条件判断来控制一个变量的值是否非空。

你也可以用空的Optional对象，对遭遇无法转换的String时返回的非法值进行建模，这时你期望parseInt的返回值是一个optional。我们无法修改最初的Java方法，但是这无碍我们进行需要的改进，你可以实现一个工具方法，将这部分逻辑封装于其中，最终返回一个我们希望的Optional对象，代码如下所示。

代码清单10-6 将String转换为Integer，并返回一个Optional对象

```
public static Optional<Integer> stringToInt(String s) {
    try {
        return Optional.of(Integer.parseInt(s));
    } catch (NumberFormatException e) {
        return Optional.empty();
    }
}
```

如果String能转换为对应的Integer，将其封装在Optioal对象中返回

否则返回一个空的Optional对象

10

我们的建议是，你可以将多个类似的方法封装到一个工具类中，让我们称之为OptionalUtility。通过这种方式，你以后就能直接调用OptionalUtility.stringToInt方法，将String转换为一个Optional<Integer>对象，而不再需要记得你在其中封装了笨拙的try/catch的逻辑了。

基础类型的Optional对象，以及为什么应该避免使用它们

不知道你注意到了没有，与Stream对象一样，Optional也提供了类似的基础类型——OptionalInt、OptionalLong以及OptionalDouble——所以代码清单10-6中的方法可以不返回Optional<Integer>，而是直接返回一个OptionalInt类型的对象。第5章中，我们讨论过使用基础类型Stream的场景，尤其是如果Stream对象包含了大量元素，出于性能的考量，使用基础类型是不错的选择，但对Optional对象而言，这个理由就不成立了，因为Optional对象最多只包含一个值。

我们不推荐大家使用基础类型的Optional，因为基础类型的Optional不支持map、

flatMap以及filter方法，而这些却是Optional类最有用的方法（正如我们在10.2节所看到的那样）。此外，与Stream一样，Optional对象无法由基础类型的Optional组合构成，所以，举例而言，如果代码清单10-6中返回的是OptionalInt类型的对象，你就不能将其作为方法引用传递给另一个Optional对象的flatMap方法。

10.4.3　把所有内容整合起来

为了展示之前介绍过的Optional类的各种方法整合在一起的威力，我们假设你需要向你的程序传递一些属性。为了举例以及测试你开发的代码，你创建了一些示例属性，如下所示：

```
Properties props = new Properties();
props.setProperty("a", "5");
props.setProperty("b", "true");
props.setProperty("c", "-3");
```

现在，我们假设你的程序需要从这些属性中读取一个值，该值是以秒为单位计量的一段时间。由于一段时间必须是正数，你想要该方法符合下面的签名：

```
public int readDuration(Properties props, String name)
```

即，如果给定属性对应的值是一个代表正整数的字符串，就返回该整数值，任何其他的情况都返回0。为了明确这些需求，你可以采用JUnit的断言，将它们形式化：

```
assertEquals(5, readDuration(param, "a"));
assertEquals(0, readDuration(param, "b"));
assertEquals(0, readDuration(param, "c"));
assertEquals(0, readDuration(param, "d"));
```

这些断言反映了初始的需求：如果属性是a，readDuration方法返回5，因为该属性对应的字符串能映射到一个正数；对于属性b，方法的返回值是0，因为它对应的值不是一个数字；对于c，方法的返回值是0，因为虽然它对应的值是个数字，不过它是个负数；对于d，方法的返回值是0，因为并不存在该名称对应的属性。让我们以命令式编程的方式实现满足这些需求的方法，代码清单如下所示。

代码清单10-7　以命令式编程的方式从属性中读取duration值

```
public int readDuration(Properties props, String name) {
    String value = props.getProperty(name);
    if (value != null) {                          // 确保名称对应的属性存在
        try {
            int i = Integer.parseInt(value);      // 将String属性转换为数字类型
            if (i > 0) {                          // 检查返回的数字是否为正数
                return i;
            }
        } catch (NumberFormatException nfe) { }
    }
    return 0;                                     // 如果前述的条件都不满足，返回0
}
```

你可能已经预见，最终的实现既复杂又不具备可读性，呈现为多个由if语句及try/catch块儿构成的嵌套条件。花几分钟时间思考一下测验10.3，想想怎样使用本章内容实现同样的效果。

测验10.3：使用Optional从属性中读取duration

请尝试使用Optional类提供的特性及代码清单10-6中提供的工具方法，通过一条精炼的语句重构代码清单10-7中的方法。

答案：如果需要访问的属性值不存在，Properties.getProperty(String)方法的返回值就是一个null，使用ofNullable工厂方法非常轻易地就能把该值转换为Optional对象。接着，你可以向它的flatMap方法传递代码清单10-6中实现的OptionalUtility.stringToInt方法的引用，将Optional<String>转换为Optional<Integer>。最后，你非常轻易地就可以过滤掉负数。这种方式下，如果任何一个操作返回一个空的Optional对象，该方法都会返回orElse方法设置的默认值0；否则就返回封装在Optional对象中的正整数。下面就是这段简化的实现：

```
public int readDuration(Properties props, String name) {
    return Optional.ofNullable(props.getProperty(name))
                   .flatMap(OptionalUtility::stringToInt)
                   .filter(i -> i > 0)
                   .orElse(0);
}
```

注意到使用Optional和Stream时的那些通用模式了吗？它们都是对数据库查询过程的反思，查询时，多种操作会被串接在一起执行。

10.5 小结

这一章中，你学到了以下的内容。

- null引用在历史上被引入到程序设计语言中，目的是为了表示变量值的缺失。
- Java 8中引入了一个新的类java.util.Optional<T>，对存在或缺失的变量值进行建模。
- 你可以使用静态工厂方法Optional.empty、Optional.of以及Optional.ofNullable创建Optional对象。
- Optional类支持多种方法，比如map、flatMap、filter，它们在概念上与Stream类中对应的方法十分相似。
- 使用Optional会迫使你更积极地解引用Optional对象，以应对变量值缺失的问题，最终，你能更有效地防止代码中出现不期而至的空指针异常。
- 使用Optional能帮助你设计更好的API，用户只需要阅读方法签名，就能了解该方法是否接受一个Optional类型的值。

第 11 章

CompletableFuture:组合式异步编程

本章内容

- 创建异步计算，并获取计算结果
- 使用非阻塞操作提升吞吐量
- 设计和实现异步API
- 如何以异步的方式使用同步的API
- 如何对两个或多个异步操作进行流水线和合并操作
- 如何处理异步操作的完成状态

最近这些年，两种趋势不断地推动我们反思我们设计软件的方式。第一种趋势和应用运行的硬件平台相关，第二种趋势与应用程序的架构相关，尤其是它们之间如何交互。我们在第7章中已经讨论过硬件平台的影响。我们注意到随着多核处理器的出现，提升应用程序处理速度最有效的方式是编写能充分发挥多核能力的软件。你已经看到通过切分大型的任务，让每个子任务并行运行，这一目标是能够实现的；你也已经了解相对直接使用线程的方式，使用分支/合并框架（在Java 7中引入）和并行流（在Java 8中新引入）能以更简单、更有效的方式实现这一目标。

第二种趋势反映在公共API日益增长的互联网服务应用。著名的互联网大鳄们纷纷提供了自己的公共API服务，比如谷歌提供了地理信息服务，Facebook提供了社交信息服务，Twitter提供了新闻服务。现在，很少有网站或者网络应用会以完全隔离的方式工作。更多的时候，我们看到的下一代网络应用都采用“混聚”（mash-up）的方式：它会使用来自多个来源的内容，将这些内容聚合在一起，方便用户的生活。

比如，你可能希望为你的法国客户提供指定主题的热点报道。为实现这一功能，你需要向谷歌或者Twitter的API请求所有语言中针对该主题最热门的评论，可能还需要依据你的内部算法对它们的相关性进行排序。之后，你可能还需要使用谷歌的翻译服务把它们翻译成法语，甚至利用谷歌地图服务定位出评论作者的位置信息，最终将所有这些信息聚集起来，呈现在你的网站上。

当然，如果某些外部网络服务发生响应慢的情况，你希望依旧能为用户提供部分信息，比如

提供带问号标记的通用地图，以文本的方式显示信息，而不是呆呆地显示一片空白屏幕，直到地图服务器返回结果或者超时退出。图11-1解释了这种典型的“混聚”应用如何与所需的远程服务交互。

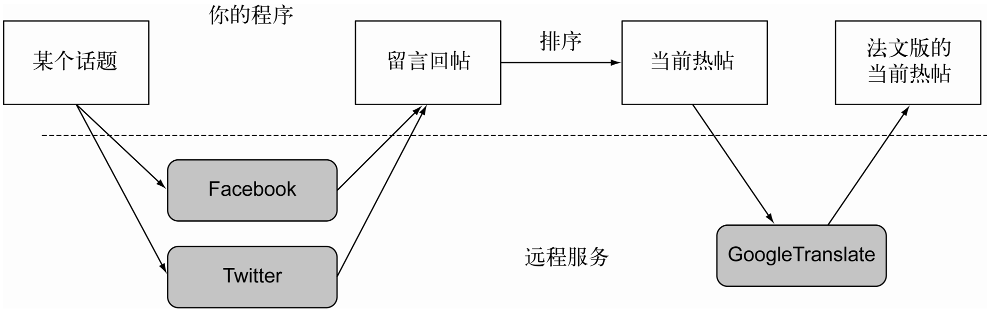


图11-1　典型的“混聚”式应用

要实现类似的服务，你需要与互联网上的多个Web服务通信。可是，你并不希望因为等待某些服务的响应，阻塞应用程序的运行，浪费数十亿宝贵的CPU时钟周期。比如，不要因为等待Facebook的数据，暂停对来自Twitter的数据处理。

这些场景体现了多任务程序设计的另一面。第7章中介绍的分支/合并框架以及并行流是实现并行处理的宝贵工具；它们将一个操作切分为多个子操作，在多个不同的核、CPU甚至是机器上并行地执行这些子操作。

与此相反，如果你的意图是实现并发，而非并行，或者你的主要目标是在同一个CPU上执行几个松耦合的任务，充分利用CPU的核，让其足够忙碌，从而最大化程序的吞吐量，那么你其实真正想做的是避免因为等待远程服务的返回，或者对数据库的查询，而阻塞线程的执行，浪费宝贵的计算资源，因为这种等待的时间很可能相当长。通过本章中你会了解，`Future`接口，尤其是它的新版实现`CompletableFuture`，是处理这种情况的利器。图11-2说明了并行和并发的区别。

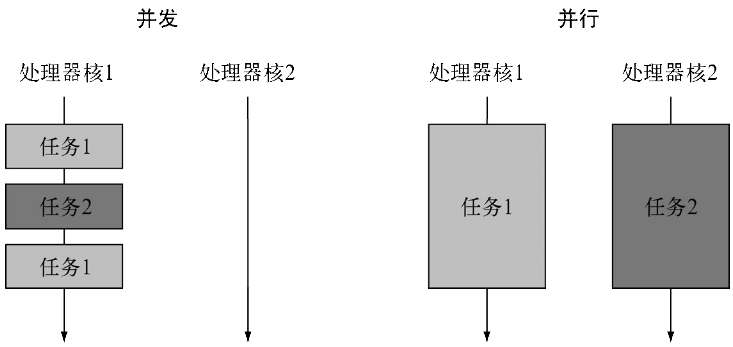


图11-2　并发和并行

11.1　Future 接口

Future接口在Java 5中被引入，设计初衷是对将来某个时刻会发生的结果进行建模。它建模了一种异步计算，返回一个执行运算结果的引用，当运算结束后，这个引用被返回给调用方。在Future中触发那些潜在耗时的操作把调用线程解放出来，让它能继续执行其他有价值的工作，不再需要呆呆等待耗时的操作完成。打个比方，你可以把它想象成这样的场景：你拿了一袋子衣服到你中意的干洗店去洗。干洗店的员工会给你张发票，告诉你什么时候你的衣服会洗好（这就是一个Future事件）。衣服干洗的同时，你可以去做其他的事情。Future的另一个优点是它比更底层的Thread更易用。要使用Future，通常你只需要将耗时的操作封装在一个Callable对象中，再将它提交给ExecutorService，就万事大吉了。下面这段代码展示了Java 8之前使用Future的一个例子。

代码清单11-1　使用Future以异步的方式执行一个耗时的操作

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<Double> future = executor.submit(new Callable<Double>() {
        public Double call() {
            return doSomeLongComputation();
        }});
doSomethingElse();

try {
    Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
    // 计算抛出一个异常
} catch (InterruptedException ie) {
    // 当前线程在等待过程中被中断
} catch (TimeoutException te) {
    // 在Future对象完成之前超过已过期
}
```

- 创建ExecutorService，通过它你可以向线程池提交任务
- 向ExecutorService提交一个Callable对象
- 以异步方式在新的线程中执行耗时的操作
- 异步操作进行的同时，你可以做其他的事情
- 获取异步操作的结果，如果最终被阻塞，无法得到结果，那么在最多等待1秒钟之后退出

正像图11-3介绍的那样，这种编程方式让你的线程可以在ExecutorService以并发方式调用另一个线程执行耗时操作的同时，去执行一些其他的任务。接着，如果你已经运行到没有异步操作的结果就无法继续任何有意义的工作时，可以调用它的get方法去获取操作的结果。如果操作已经完成，该方法会立刻返回操作的结果，否则它会阻塞你的线程，直到操作完成，返回相应的结果。

你能想象这种场景存在怎样的问题吗？如果该长时间运行的操作永远不返回了会怎样？为了处理这种可能性，虽然Future提供了一个无需任何参数的get方法，我们还是推荐大家使用重载版本的get方法，它接受一个超时的参数，通过它，你可以定义你的线程等待Future结果的最长时间，而不是像代码清单11-1中那样永无止境地等待下去。

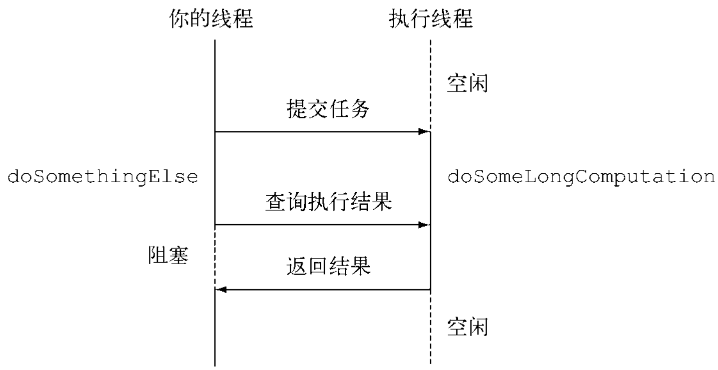


图11-3　使用Future以异步方式执行长时间的操作

11.1.1　Future 接口的局限性

通过第一个例子，我们知道Future接口提供了方法来检测异步计算是否已经结束（使用isDone方法），等待异步操作结束，以及获取计算的结果。但是这些特性还不足以让你编写简洁的并发代码。比如，我们很难表述Future结果之间的依赖性；从文字描述上这很简单，“当长时间计算任务完成时，请将该计算的结果通知到另一个长时间运行的计算任务，这两个计算任务都完成后，将计算的结果与另一个查询操作结果合并”。但是，使用Future中提供的方法完成这样的操作又是另外一回事。这也是我们需要更具描述能力的特性的原因，比如下面这些。

- 将两个异步计算合并为一个——这两个异步计算之间相互独立，同时第二个又依赖于第一个的结果。
- 等待Future集合中的所有任务都完成。
- 仅等待Future集合中最快结束的任务完成（有可能因为它们试图通过不同的方式计算同一个值），并返回它的结果。
- 通过编程方式完成一个Future任务的执行（即以手工设定异步操作结果的方式）。
- 应对Future的完成事件（即当Future的完成事件发生时会收到通知，并能使用Future计算的结果进行下一步的操作，不只是简单地阻塞等待操作的结果）。

这一章中，你会了解新的CompletableFuture类（它实现了Future接口）如何利用Java 8的新特性以更直观的方式将上述需求都变为可能。Stream和CompletableFuture的设计都遵循了类似的模式：它们都使用了Lambda表达式以及流水线的思想。从这个角度，你可以说CompletableFuture和Future的关系就跟Stream和Collection的关系一样。

11.1.2　使用 CompletableFuture 构建异步应用

为了展示CompletableFuture的强大特性，我们会创建一个名为“最佳价格查询器”（best-price-finder）的应用，它会查询多个在线商店，依据给定的产品或服务找出最低的价格。这

个过程中，你会学到几个重要的技能。

- 首先，你会学到如何为你的客户提供异步API（如果你拥有一间在线商店的话，这是非常有帮助的）。
- 其次，你会掌握如何让你使用了同步API的代码变为非阻塞代码。你会了解如何使用流水线将两个接续的异步操作合并为一个异步计算操作。这种情况肯定会出现，比如，在线商店返回了你想要购买商品的原始价格，并附带着一个折扣代码——最终，要计算出该商品的实际价格，你不得不访问第二个远程折扣服务，查询该折扣代码对应的折扣比率。
- 你还会学到如何以响应式的方式处理异步操作的完成事件，以及随着各个商店返回它的商品价格，最佳价格查询器如何持续地更新每种商品的最佳推荐，而不是等待所有的商店都返回他们各自的价格（这种方式存在着一定的风险，一旦某家商店的服务中断，用户可能遭遇白屏）。

同步API与异步API

同步API其实只是对传统方法调用的另一种称呼：你调用了某个方法，调用方在被调用方运行的过程中会等待，被调用方运行结束返回，调用方取得被调用方的返回值并继续运行。即使调用方和被调用方在不同的线程中运行，调用方还是需要等待被调用方结束运行，这就是**阻塞式调用**这个名词的由来。

与此相反，**异步API**会直接返回，或者至少在被调用方计算完成之前，将它剩余的计算任务交给另一个线程去做，该线程和调用方是异步的——这就是**非阻塞式调用**的由来。执行剩余计算任务的线程会将它的计算结果返回给调用方。返回的方式要么是通过回调函数，要么是由调用方再次执行一个“等待，直到计算完成”的方法调用。这种方式的计算在I/O系统程序设计中非常常见：你发起了一次磁盘访问，这次访问和你的其他计算操作是异步的，你完成其他的任务时，磁盘块的数据可能还没载入到内存，你只需要等待数据的载入完成。

11.2　实现异步 API

为了实现最佳价格查询器应用，让我们从每个商店都应该提供的API定义入手。首先，商店应该声明依据指定产品名称返回价格的方法：

```
public class Shop {
    public double getPrice(String product) {
        // 待实现
    }
}
```

该方法的内部实现会查询商店的数据库，但也有可能执行一些其他耗时的任务，比如联系其他外部服务（比如，商店的供应商，或者跟制造商相关的推广折扣）。我们在本章剩下的内容中，采用`delay`方法模拟这些长期运行的方法的执行，它会人为地引入1秒钟的延迟，方法声明如下。

代码清单11-2 模拟1秒钟延迟的方法

```
public static void delay() {
    try {
        Thread.sleep(1000L);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

为了介绍本章的内容，getPrice方法会调用delay方法，并返回一个随机计算的值，代码清单如下所示。返回随机计算的价格这段代码看起来有些取巧。它使用charAt，依据产品的名称，生成一个随机值作为价格。

代码清单11-3 在getPrice方法中引入一个模拟的延迟

```
public double getPrice(String product) {
    return calculatePrice(product);
}
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

很明显，这个API的使用者（这个例子中为最佳价格查询器）调用该方法时，它依旧会被阻塞。为等待同步事件完成而等待1秒钟，这是无法接受的，尤其是考虑到最佳价格查询器对网络中的所有商店都要重复这种操作。本章接下来的小节中，你会了解如何以异步方式使用同步API解决这个问题。但是，出于学习如何设计异步API的考虑，我们会继续这一节的内容，假装我们还在深受这一困难的烦扰：你是一个睿智的商店店主，你已经意识到了这种同步API会为你的用户带来多么痛苦的体验，你希望以异步API的方式重写这段代码，让用户更流畅地访问你的网站。

11.2.1 将同步方法转换为异步方法

为了实现这个目标，你首先需要将getPrice转换为getPriceAsync方法，并修改它的返回值：

```
public Future<Double> getPriceAsync(String product) { ... }
```

我们在本章开头已经提到，Java 5引入了java.util.concurrent.Future接口表示一个异步计算（即调用线程可以继续运行，不会因为调用方法而阻塞）的结果。这意味着Future是一个暂时还不可知值的处理器，这个值在计算完成后，可以通过调用它的get方法取得。因为这样的设计，getPriceAsync方法才能立刻返回，给调用线程一个机会，能在同一时间去执行其他有价值的计算任务。新的CompletableFuture类提供了大量的方法，让我们有机会以多种可能的方式轻松地实现这个方法，比如下面就是这样一段实现代码。

代码清单11-4　getPriceAsync方法的实现

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread( () -> {
                double price = calculatePrice(product);
                futurePrice.complete(price);
    }).start();
    return futurePrice;
}
```

在另一个线程中以异步方式执行计算

创建CompletableFuture对象，它会包含计算的结果

需长时间计算的任务结束并得出结果时，设置Future的返回值

无需等待还没结束的计算，直接返回Future对象

在这段代码中，你创建了一个代表异步计算的CompletableFuture对象实例，它在计算完成时会包含计算的结果。接着，你调用fork创建了另一个线程去执行实际的价格计算工作，不等该耗时计算任务结束，直接返回一个Future实例。当请求的产品价格最终计算得出时，你可以使用它的complete方法，结束completableFuture对象的运行，并设置变量的值。很显然，这个新版Future的名称也解释了它所具有的特性。使用这个API的客户端，可以通过下面的这段代码对其进行调用。

代码清单11-5　使用异步API

```
Shop shop = new Shop("BestShop");
long start = System.nanoTime();
Future<Double> futurePrice = shop.getPriceAsync("my favorite product");
long invocationTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Invocation returned after " + invocationTime
                                                + " msecs");
// 执行更多任务，比如查询其他商店
doSomethingElse();
// 在计算商品价格的同时
try {
    double price = futurePrice.get();
    System.out.printf("Price is %.2f%n", price);
} catch (Exception e) {
    throw new RuntimeException(e);
}
long retrievalTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Price returned after " + retrievalTime + " msecs");
```

查询商店，试图取得商品的价格

从Future对象中读取价格，如果价格未知，会发生阻塞

我们看到这段代码中，客户向商店查询了某种商品的价格。由于商店提供了异步API，该次调用立刻返回了一个Future对象，通过该对象客户可以在将来的某个时刻取得商品的价格。这种方式下，客户在进行商品价格查询的同时，还能执行一些其他的任务，比如查询其他家商店中商品的价格，不会呆呆地阻塞在那里等待第一家商店返回请求的结果。最后，如果所有有意义的工作都已经完成，客户所有要执行的工作都依赖于商品价格时，再调用Future的get方法。执行了这个操作后，客户要么获得Future中封装的值（如果异步任务已经完成），要么发生阻塞，直到该异步任务完成，期望的值能够访问。代码清单11-5产生的输出可能是下面这样：

```
Invocation returned after 43 msecs
Price is 123.26
Price returned after 1045 msecs
```

你一定已经发现getPriceAsync方法的调用返回远远早于最终价格计算完成的时间。在11.4节中，你还会知道我们有可能避免发生客户端被阻塞的风险。实际上这非常简单，Future执行完毕可以发送一个通知，仅在计算结果可用时执行一个由Lambda表达式或者方法引用定义的回调函数。不过，我们当下不会对此进行讨论，现在我们要解决的是另一个问题：如何正确地管理异步任务执行过程中可能出现的错误。

11.2.2 错误处理

如果没有意外，我们目前开发的代码工作得很正常。但是，如果价格计算过程中产生了错误会怎样呢？非常不幸，这种情况下你会得到一个相当糟糕的结果：用于提示错误的异常会被限制在试图计算商品价格的当前线程的范围内，最终会杀死该线程，而这会导致等待get方法返回结果的客户端永久地被阻塞。

客户端可以使用重载版本的get方法，它使用一个超时参数来避免发生这样的情况。这是一种值得推荐的做法，你应该尽量在你的代码中添加超时判断的逻辑，避免发生类似的问题。使用这种方法至少能防止程序永久地等待下去，超时发生时，程序会得到通知发生了TimeoutException。不过，也因为如此，你不会有机会发现计算商品价格的线程内到底发生了什么问题才引发了这样的失效。为了让客户端能了解商店无法提供请求商品价格的原因，你需要使用CompletableFuture的completeExceptionally方法将导致CompletableFuture内发生问题的异常抛出。对代码清单11-4优化后的结果如下所示。

代码清单11-6 抛出CompletableFuture内的异常

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread( () -> {
                try {
                    double price = calculatePrice(product);
                    futurePrice.complete(price);
                } catch (Exception ex) {
                    futurePrice.completeExceptionally(ex);
                }
    }).start();
    return futurePrice;
}
```

如果价格计算正常结束，完成**Future**操作并设置商品价格

否则就抛出导致失败的异常，完成这次**Future**操作

客户端现在会收到一个ExecutionException异常，该异常接收了一个包含失败原因的Exception参数，即价格计算方法最初抛出的异常。所以，举例来说，如果该方法抛出了一个运行时异常“product not available”，客户端就会得到像下面这样一段ExecutionException：

```
java.util.concurrent.ExecutionException: java.lang.RuntimeException: product
     not available
    at java.util.concurrent.CompletableFuture.get(CompletableFuture.java:2237)
```

```
    at lambdasinaction.chap11.AsyncShopClient.main(AsyncShopClient.java:14)
    ... 5 more
Caused by: java.lang.RuntimeException: product not available
    at lambdasinaction.chap11.AsyncShop.calculatePrice(AsyncShop.java:36)
    at lambdasinaction.chap11.AsyncShop.lambda$getPrice$0(AsyncShop.java:23)
    at lambdasinaction.chap11.AsyncShop$$Lambda$1/24071475.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:744)
```

使用工厂方法supplyAsync创建CompletableFuture

目前为止我们已经了解了如何通过编程创建CompletableFuture对象以及如何获取返回值，虽然看起来这些操作已经比较方便，但还有进一步提升的空间，CompletableFuture类自身提供了大量精巧的工厂方法，使用这些方法能更容易地完成整个流程，还不用担心实现的细节。比如，采用supplyAsync方法后，你可以用一行语句重写代码清单11-4中的getPriceAsync方法，如下所示。

代码清单11-7　使用工厂方法supplyAsync创建CompletableFuture对象

```
public Future<Double> getPriceAsync(String product) {
    return CompletableFuture.supplyAsync(() -> calculatePrice(product));
}
```

supplyAsync方法接受一个生产者（Supplier）作为参数，返回一个CompletableFuture对象，该对象完成异步执行后会读取调用生产者方法的返回值。生产者方法会交由ForkJoinPool池中的某个执行线程（Executor）运行，但是你也可以使用supplyAsync方法的重载版本，传递第二个参数指定不同的执行线程执行生产者方法。一般而言，向CompletableFuture的工厂方法传递可选参数，指定生产者方法的执行线程是可行的，在11.3.4节中，你会使用这一能力，我们会在该小节介绍如何使用适合你应用特性的执行线程改善程序的性能。

此外，代码清单11-7中getPriceAsync方法返回的CompletableFuture对象和代码清单11-6中你手工创建和完成的CompletableFuture对象是完全等价的，这意味着它提供了同样的错误管理机制，而前者你花费了大量的精力才得以构建。

本章的剩余部分中，我们会假设你非常不幸，无法控制Shop类提供API的具体实现，最终提供给你的API都是同步阻塞式的方法。这也是当你试图使用服务提供的HTTP API时最常发生的情况。你会学到如何以异步的方式查询多个商店，避免被单一的请求所阻塞，并由此提升你的“最佳价格查询器”的性能和吞吐量。

11.3　让你的代码免受阻塞之苦

所以，你已经被要求进行“最佳价格查询器”应用的开发了，不过你需要查询的所有商店都如11.2节开始时介绍的那样，只提供了同步API。换句话说，你有一个商家的列表，如下所示：

```
List<Shop> shops = Arrays.asList(new Shop("BestPrice"),
                                 new Shop("LetsSaveBig"),
                                 new Shop("MyFavoriteShop"),
                                 new Shop("BuyItAll"));
```

你需要使用下面这样的签名实现一个方法，它接受产品名作为参数，返回一个字符串列表，这个字符串列表中包括商店的名称、该商店中指定商品的价格：

```
public List<String> findPrices(String product);
```

你的第一个想法可能是使用我们在第4、5、6章中学习的Stream特性。你可能试图写出类似下面这个清单中的代码（是的，作为第一个方案，如果你想到这些已经相当棒了！）。

代码清单11-8 采用顺序查询所有商店的方式实现的findPrices方法

```
public List<String> findPrices(String product) {
    return shops.stream()
        .map(shop -> String.format("%s price is %.2f",
                                   shop.getName(), shop.getPrice(product)))
        .collect(toList());
}
```

好吧，这段代码看起来非常直白。现在试着用该方法去查询你最近这些天疯狂着迷的唯一产品（是的，你已经猜到了，它就是myPhone27S）。此外，也请记录下方法的执行时间，通过这些数据，我们可以比较优化之后的方法会带来多大的性能提升，具体的代码清单如下。

代码清单11-9 验证findPrices的正确性和执行性能

```
long start = System.nanoTime();
System.out.println(findPrices("myPhone27S"));
long duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Done in " + duration + " msecs");
```

代码清单11-9的运行结果输出如下：

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 4032 msecs
```

正如你预期的，findPrices方法的执行时间仅比4秒钟多了那么几毫秒，因为对这4个商店的查询是顺序进行的，并且一个查询操作会阻塞另一个，每一个操作都要花费大约1秒左右的时间计算请求商品的价格。你怎样才能改进这个结果呢？

11.3.1 使用并行流对请求进行并行操作

读完第7章，你应该想到的第一个，可能也是最快的改善方法是使用并行流来避免顺序计算，如下所示。

代码清单11-10 对findPrices进行并行操作

```
public List<String> findPrices(String product) {
    return shops.parallelStream()
        .map(shop -> String.format("%s price is %.2f",
                                   shop.getName(), shop.getPrice(product)))
        .collect(toList());
}
```

使用并行流并行地从不同的商店获取价格

运行代码，与代码清单11-9的执行结果相比较，你发现了新版findPrices的改进了吧。

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 1180 msecs
```

相当不错啊！看起来这是个简单但有效的主意：现在对四个不同商店的查询实现了并行，所以完成所有操作的总耗时只有1秒多一点儿。你能做得更好吗？让我们尝试使用刚学过的CompletableFuture，将findPrices方法中对不同商店的同步调用替换为异步调用。

11.3.2　使用 CompletableFuture 发起异步请求

你已经知道我们可以使用工厂方法supplyAsync创建CompletableFuture对象。让我们把它利用起来：

```
List<CompletableFuture<String>> priceFutures =
        shops.stream()
        .map(shop -> CompletableFuture.supplyAsync(
             () -> String.format("%s price is %.2f",
             shop.getName(), shop.getPrice(product))))
        .collect(toList());
```

使用这种方式，你会得到一个List<CompletableFuture<String>>，列表中的每个CompletableFuture对象在计算完成后都包含商店的String类型的名称。但是，由于你用CompletableFutures实现的findPrices方法要求返回一个List<String>，你需要等待所有的future执行完毕，将其包含的值抽取出来，填充到列表中才能返回。

为了实现这个效果，你可以向最初的List<CompletableFuture<String>>施加第二个map操作，对List中的所有future对象执行join操作，一个接一个地等待它们运行结束。注意CompletableFuture类中的join方法和Future接口中的get有相同的含义，并且也声明在Future接口中，它们唯一的不同是join不会抛出任何检测到的异常。使用它你不再需要使用try/catch语句块让你传递给第二个map方法的Lambda表达式变得过于臃肿。所有这些整合在一起，你就可以重新实现findPrices了，具体代码如下。

代码清单11-11　使用CompletableFuture实现findPrices方法

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
            shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(      <-- 使用CompletableFuture以异步方式计算每种商品的价格
                         () -> shop.getName() + " price is " +
                               shop.getPrice(product)))
            .collect(Collectors.toList());

    return priceFutures.stream()
            .map(CompletableFuture::join)      <-- 等待所有异步操作结束
            .collect(toList());
}
```

注意到了吗？这里使用了两个不同的Stream流水线，而不是在同一个处理流的流水线上一个接一个地放置两个map操作——这其实是有缘由的。考虑流操作之间的延迟特性，如果你在单一流水线中处理流，发向不同商家的请求只能以同步、顺序执行的方式才会成功。因此，每个创建CompletableFuture对象只能在前一个操作结束之后执行查询指定商家的动作、通知join方法返回计算结果。图11-4解释了这些重要的细节。

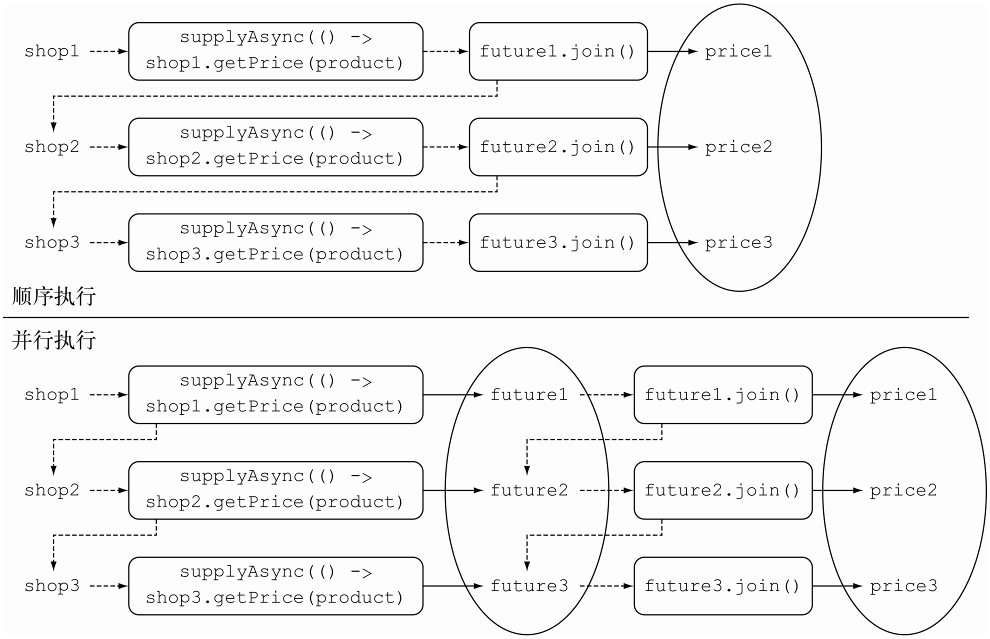


图11-4　为什么Stream的延迟特性会引起顺序执行，以及如何避免

图11-4的上半部分展示了使用单一流水线处理流的过程，我们看到，执行的流程（以虚线标识）是顺序的。事实上，新的CompletableFuture对象只有在前一个操作完全结束之后，才能创建。与此相反，图的下半部分展示了如何先将CompletableFutures对象聚集到一个列表中（即图中以椭圆表示的部分），让对象们可以在等待其他对象完成操作之前就能启动。

运行代码清单11-11中的代码来了解下第三个版本findPrices方法的性能，你会得到下面这几行输出：

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 2005 msecs
```

这个结果让人相当失望，不是吗？超过2秒意味着利用CompletableFuture实现的版本，比刚开始代码清单11-8中原生顺序执行且会发生阻塞的版本快。但是它的用时也差不多是使用并行流的前一个版本的两倍。尤其是，考虑到从顺序执行的版本转换到并行流的版本只做了非常小的改动，就让人更加沮丧。

与此形成鲜明对比的是，我们为采用CompletableFutures完成的新版方法做了大量的工作！但，这就是全部的真相吗？这种场景下使用CompletableFutures真的是浪费时间吗？或者我们可能漏掉了某些重要的东西？继续往下探究之前，让我们休息几分钟，尤其是想想你测试代码的机器是否足以以并行方式运行四个线程。①

11.3.3　寻找更好的方案

并行流的版本工作得非常好，那是因为它能并行地执行四个任务，所以它几乎能为每个商家分配一个线程。但是，如果你想要增加第五个商家到商店列表中，让你的“最佳价格查询”应用对其进行处理，这时会发生什么情况？毫不意外，顺序执行版本的执行还是需要大约5秒多钟的时间，下面是执行的输出：

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 176.08]
Done in 5025 msecs
```

使用顺序流方式的程序输出

非常不幸，并行流版本的程序这次比之前也多消耗了差不多1秒钟的时间，因为可以并行运行（通用线程池中处于可用状态的）的四个线程现在都处于繁忙状态，都在对前4个商店进行查询。第五个查询只能等到前面某一个操作完成释放出空闲线程才能继续，它的运行结果如下：

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 176.08]
Done in 2177 msecs
```

使用并行流方式的程序输出

CompletableFuture版本的程序结果如何呢？我们也试着添加第5个商店对其进行了测试，结果如下：

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 176.08]
Done in 2006 msecs
```

使用CompletableFuture的程序输出

CompletableFuture版本的程序似乎比并行流版本的程序还快那么一点儿。但是最后这个版本也不太令人满意。比如，如果你试图让你的代码处理9个商店，并行流版本耗时3143毫秒，而CompletableFuture版本耗时3009毫秒。它们看起来不相伯仲，究其原因都一样：它们内部采用的是同样的通用线程池，默认都使用固定数目的线程，具体线程数取决于Runtime.getRuntime().availableProcessors()的返回值。然而，CompletableFuture具有一定的优势，因为它允许你对执行器（Executor）进行配置，尤其是线程池的大小，让它以更适合应

① 如果你使用的机器足够强大，能以并行方式运行更多的线程（比如说8个线程），那你需要使用更多的商店和并行进程，才能重现这几页中介绍的行为。

用需求的方式进行配置，满足程序的要求，而这是并行流API无法提供的。让我们看看你怎样利用这种配置上的灵活性带来实际应用程序性能上的提升。

11.3.4　使用定制的执行器

就这个主题而言，明智的选择似乎是创建一个配有线程池的执行器，线程池中线程的数目取决于你预计你的应用需要处理的负荷，但是你该如何选择合适的线程数目呢？

调整线程池的大小

《Java并发编程实战》（http://mng.bz/979c）一书中，Brian Goetz和合著者们为线程池大小的优化提供了不少中肯的建议。这非常重要，如果线程池中线程的数量过多，最终它们会竞争稀缺的处理器和内存资源，浪费大量的时间在上下文切换上。反之，如果线程的数目过少，正如你的应用所面临的情况，处理器的一些核可能就无法充分利用。Brian Goetz建议，线程池大小与处理器的利用率之比可以使用下面的公式进行估算：

$$N_{threads} = N_{CPU} * U_{CPU} * (1 + W/C)$$

其中：

- ❑ N_{CPU}是处理器的核的数目，可以通过`Runtime.getRuntime().availableProcessors()`得到
- ❑ U_{CPU}是期望的CPU利用率（该值应该介于0和1之间）
- ❑ W/C是等待时间与计算时间的比率

你的应用99%的时间都在等待商店的响应，所以估算出的W/C比率为100。这意味着如果你期望的CPU利用率是100%，你需要创建一个拥有400个线程的线程池。实际操作中，如果你创建的线程数比商店的数目更多，反而是一种浪费，因为这样做之后，你线程池中的有些线程根本没有机会被使用。出于这种考虑，我们建议你将执行器使用的线程数，与你需要查询的商店数目设定为同一个值，这样每个商店都应该对应一个服务线程。不过，为了避免发生由于商店的数目过多导致服务器超负荷而崩溃，你还是需要设置一个上限，比如100个线程。代码清单如下所示。

11

代码清单11-12　为“最优价格查询器”应用定制的执行器

```
private final Executor executor =
        Executors.newFixedThreadPool(Math.min(shops.size(), 100),
                                     new ThreadFactory() {
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setDaemon(true);
                return t;
            }
});
```

创建一个线程池，线程池中线程的数目为100和商店数目二者中较小的一个值

使用守护线程——这种方式不会阻止程序的关停

注意，你现在正创建的是一个由守护线程构成的线程池。Java程序无法终止或者退出一个正在运行中的线程，所以最后剩下的那个线程会由于一直等待无法发生的事件而引发问题。与此相反，如果将线程标记为守护进程，意味着程序退出时它也会被回收。这二者之间没有性能上的差异。现在，你可以将执行器作为第二个参数传递给supplyAsync工厂方法了。比如，你现在可以按照下面的方式创建一个可查询指定商品价格的CompletableFuture对象：

```
CompletableFuture.supplyAsync(() -> shop.getName() + " price is " +
                                    shop.getPrice(product), executor);
```

改进之后，使用CompletableFuture方案的程序处理5个商店仅耗时1021秒，处理9个商店时耗时1022秒。一般而言，这种状态会一直持续，直到商店的数目达到我们之前计算的阈值400。这个例子证明了要创建更适合你的应用特性的执行器，利用CompletableFutures向其提交任务执行是个不错的主意。处理需大量使用异步操作的情况时，这几乎是最有效的策略。

并行——使用流还是CompletableFutures？

目前为止，你已经知道对集合进行并行计算有两种方式：要么将其转化为并行流，利用map这样的操作开展工作，要么枚举出集合中的每一个元素，创建新的线程，在CompletableFuture内对其进行操作。后者提供了更多的灵活性，你可以调整线程池的大小，而这能帮助你确保整体的计算不会因为线程都在等待I/O而发生阻塞。

我们对使用这些API的建议如下。

- 如果你进行的是计算密集型的操作，并且没有I/O，那么推荐使用Stream接口，因为实现简单，同时效率也可能是最高的（如果所有的线程都是计算密集型的，那就没有必要创建比处理器核数更多的线程）。
- 反之，如果你并行的工作单元还涉及等待I/O的操作（包括网络连接等待），那么使用CompletableFuture灵活性更好，你可以像前文讨论的那样，依据等待/计算，或者W/C的比率设定需要使用的线程数。这种情况不使用并行流的另一个原因是，处理流的流水线中如果发生I/O等待，流的延迟特性会让我们很难判断到底什么时候触发了等待。

现在你已经了解了如何利用CompletableFuture为你的用户提供异步API，以及如何将一个同步又缓慢的服务转换为异步的服务。不过到目前为止，我们每个Future中进行的都是单次的操作。下一节中，你会看到如何将多个异步操作结合在一起，以流水线的方式运行，从描述形式上，它与你在前面学习的Stream API有几分类似。

11.4　对多个异步任务进行流水线操作

让我们假设所有的商店都同意使用一个集中式的折扣服务。该折扣服务提供了五个不同的折扣代码，每个折扣代码对应不同的折扣率。你使用一个枚举型变量Discount.Code来实现这一想法，具体代码如下所示。

代码清单11-13　以枚举类型定义的折扣代码

```
public class Discount {
    public enum Code {
        NONE(0), SILVER(5), GOLD(10), PLATINUM(15), DIAMOND(20);

        private final int percentage;

        Code(int percentage) {
            this.percentage = percentage;
        }
    }
    // Discount类的具体实现这里暂且不表示，参见代码清单11-14
}
```

我们还假设所有的商店都同意修改getPrice方法的返回格式。getPrice现在以ShopName:price:DiscountCode的格式返回一个String类型的值。我们的示例实现中会返回一个随机生成的Discount.Code，以及已经计算得出的随机价格：

```
public String getPrice(String product) {
    double price = calculatePrice(product);
    Discount.Code code = Discount.Code.values()[
                            random.nextInt(Discount.Code.values().length)];
    return String.format("%s:%.2f:%s", name, price, code);
}
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

调用getPrice方法可能会返回像下面这样一个String值：

```
BestPrice:123.26:GOLD
```

11.4.1　实现折扣服务

你的“最佳价格查询器”应用现在能从不同的商店取得商品价格，解析结果字符串，针对每个字符串，查询折扣服务取的折扣代码[①]。这个流程决定了请求商品的最终折扣价格（每个折扣代码的实际折扣比率有可能发生变化，所以你每次都需要查询折扣服务）。我们已经将对商店返回字符串的解析操作封装到了下面的Quote类之中：

```
public class Quote {

    private final String shopName;
    private final double price;
    private final Discount.Code discountCode;

    public Quote(String shopName, double price, Discount.Code code) {
        this.shopName = shopName;
```

① 原文为for each String, query the discount server’s needs，此处在上下文中略有不通，疑为原文有误。——译者注

```
        this.price = price;
        this.discountCode = code;
    }

    public static Quote parse(String s) {
        String[] split = s.split(":");
        String shopName = split[0];
        double price = Double.parseDouble(split[1]);
        Discount.Code discountCode = Discount.Code.valueOf(split[2]);
        return new Quote(shopName, price, discountCode);
    }

    public String getShopName() { return shopName; }
    public double getPrice() { return price; }
    public Discount.Code getDiscountCode() { return discountCode; }
}
```

通过传递shop对象返回的字符串给静态工厂方法parse，你可以得到Quote类的一个实例，它包含了shop的名称、折扣之前的价格，以及折扣代码。

Discount服务还提供了一个applyDiscount方法，它接收一个Quote对象，返回一个字符串，表示生成该Quote的shop中的折扣价格，代码如下所示。

代码清单11-14　Discount服务

```
public class Discount {
    public enum Code {
        // 源码暂时省略……
    }

    public static String applyDiscount(Quote quote) {
        return quote.getShopName() + " price is " +
               Discount.apply(quote.getPrice(),          <-- 将折扣代码应用于商品最初的原始价格
                              quote.getDiscountCode());
    }
    private static double apply(double price, Code code) {
        delay();                                          <-- 模拟 Discount 服务响应的延迟
        return format(price * (100 - code.percentage) / 100);
    }
}
```

11.4.2　使用 Discount 服务

由于Discount服务是一种远程服务，你还需要增加1秒钟的模拟延迟，代码如下所示。和在11.3节中一样，首先尝试以最直接的方式（坏消息是，这种方式是顺序而且同步执行的）重新实现findPrices，以满足这些新增的需求。

代码清单11-15　以最简单的方式实现使用Discount服务的findPrices方法

```
public List<String> findPrices(String product) {
    return shops.stream()
            .map(shop -> shop.getPrice(product))      <-- 取得每个shop对象中商品的原始价格
```

```
                .map(Quote::parse)                    <—— 在Quote对象中对shop返回的字符串进行转换
                .map(Discount::applyDiscount)         <—— 联系Discount服务，为每个Quote申请折扣
                .collect(toList());
}
```

通过在shop构成的流上采用流水线方式执行三次map操作，我们得到了期望的结果。

- 第一个操作将每个shop对象转换成了一个字符串，该字符串包含了该 shop中指定商品的价格和折扣代码。
- 第二个操作对这些字符串进行了解析，在Quote对象中对它们进行转换。
- 最终，第三个map会操作联系远程的Discount服务，计算出最终的折扣价格，并返回该价格及提供该价格商品的shop。

你可能已经猜到，这种实现方式的性能远非最优，不过我们还是应该测量一下。跟之前一样，通过运行基准测试，我们得到下面的数据：

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 10028 msecs
```

毫无意外，这次执行耗时10秒，因为顺序查询5个商店耗时大约5秒，现在又加上了Discount服务为5个商店返回的价格申请折扣所消耗的5秒钟。你已经知道，把流转换为并行流的方式，非常容易提升该程序的性能。不过，通过11.3节的介绍，你也知道这一方案在商店的数目增加时，扩展性不好，因为Stream底层依赖的是线程数量固定的通用线程池。相反，你也知道，如果自定义CompletableFutures调度任务执行的执行器能够更充分地利用CPU资源。

11.4.3　构造同步和异步操作

让我们再次使用CompletableFuture提供的特性，以异步方式重新实现findPrices方法。详细代码如下所示。如果你发现有些内容不太熟悉，不用太担心，我们很快会进行针对性的介绍。

代码清单11-16　使用CompletableFuture实现findPrices方法

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
        shops.stream()
             .map(shop -> CompletableFuture.supplyAsync(          <—— 以异步方式取得每个shop中指定产品的原始价格
                                   () -> shop.getPrice(product), executor))
             .map(future -> future.thenApply(Quote::parse))        <—— Quote对象存在时，对其返回的值进行转换
             .map(future -> future.thenCompose(quote ->            <—— 使用另一个异步任务构造期望的Future，申请折扣
                         CompletableFuture.supplyAsync(
                           () -> Discount.applyDiscount(quote), executor)))
                .collect(toList());

    return priceFutures.stream()
            .map(CompletableFuture::join)      <—— 等待流中的所有Future执行完毕，并提取各自的返回值
            .collect(toList());
}
```

这一次，事情看起来变得更加复杂了，所以让我们一步一步地理解到底发生了什么。这三次转换的流程如图11-5所示。

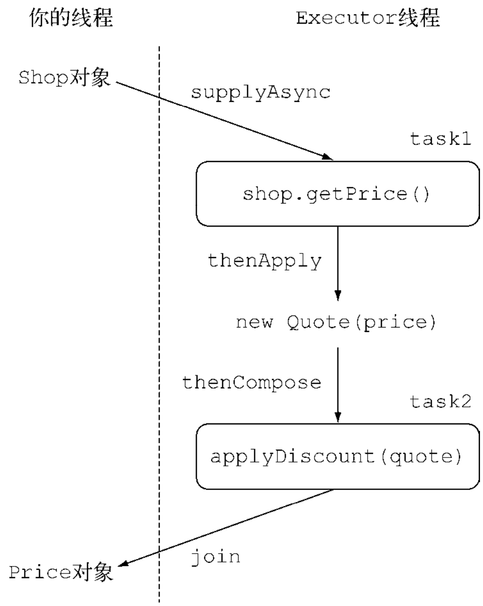


图11-5　构造同步操作和异步任务

你所进行的这三次map操作和代码清单11-5中的同步方案没有太大的区别，不过你使用CompletableFuture类提供的特性，在需要的地方把它们变成了异步操作。

1. 获取价格

这三个操作中的第一个你已经在本章的各个例子中见过很多次，只需要将Lambda表达式作为参数传递给supplyAsync工厂方法就可以以异步方式对shop进行查询。第一个转换的结果是一个Stream<CompletableFuture<String>>，一旦运行结束，每个CompletableFuture对象中都会包含对应shop返回的字符串。注意，你对CompletableFuture进行了设置，用代码清单11-12中的方法向其传递了一个订制的执行器Executor。

2. 解析报价

现在你需要进行第二次转换将字符串转变为订单。由于一般情况下解析操作不涉及任何远程服务，也不会进行任何I/O操作，它几乎可以在第一时间进行，所以能够采用同步操作，不会带来太多的延迟。由于这个原因，你可以对第一步中生成的CompletableFuture对象调用它的thenApply，将一个由字符串转换Quote的方法作为参数传递给它。

注意到了吗？直到你调用的CompletableFuture执行结束，使用的thenApply方法都不会阻塞你代码的执行。这意味着CompletableFuture最终结束运行时，你希望传递Lambda表达式给thenApply方法，将Stream中的每个CompletableFuture<String>对象转换为对应的CompletableFuture<Quote>对象。你可以把这看成是为处理CompletableFuture的结果建

立了一个菜单，就像你曾经为Stream的流水线所做的事儿一样。

3. 为计算折扣价格构造Future

第三个map操作涉及联系远程的Discount服务，为从商店中得到的原始价格申请折扣率。这一转换与前一个转换又不大一样，因为这一转换需要远程执行（或者，就这个例子而言，它需要模拟远程调用带来的延迟），出于这一原因，你也希望它能够异步执行。

为了实现这一目标，你像第一个调用传递getPrice给supplyAsync那样，将这一操作以Lambda表达式的方式传递给了supplyAsync工厂方法，该方法最终会返回另一个CompletableFuture对象。到目前为止，你已经进行了两次异步操作，用了两个不同的CompletableFutures对象进行建模，你希望能把它们以级联的方式串接起来进行工作。

- 从shop对象中获取价格，接着把价格转换为Quote。
- 拿到返回的Quote对象，将其作为参数传递给Discount服务，取得最终的折扣价格。

Java 8的 CompletableFuture API提供了名为thenCompose的方法，它就是专门为这一目的而设计的，thenCompose方法允许你对两个异步操作进行流水线，第一个操作完成时，将其结果作为参数传递给第二个操作。换句话说，你可以创建两个CompletableFutures对象，对第一个CompletableFuture对象调用thenCompose，并向其传递一个函数。当第一个CompletableFuture执行完毕后，它的结果将作为该函数的参数，这个函数的返回值是以第一个CompletableFuture的返回做输入计算出的第二个CompletableFuture对象。使用这种方式，即使Future在向不同的商店收集报价，主线程还是能继续执行其他重要的操作，比如响应UI事件。

将这三次map操作的返回的Stream元素收集到一个列表，你就得到了一个List<CompletableFuture<String>>，等这些CompletableFuture对象最终执行完毕，你就可以像代码清单11-11中那样利用join取得它们的返回值。代码清单11-18实现的新版findPrices方法产生的输出如下：

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 2035 msecs
```

11

你在代码清单11-16中使用的thenCompose方法像CompletableFuture类中的其他方法一样，也提供了一个以Async后缀结尾的版本thenComposeAsync。通常而言，名称中不带Async的方法和它的前一个任务一样，在同一个线程中运行；而名称以Async结尾的方法会将后续的任务提交到一个线程池，所以每个任务是由不同的线程处理的。就这个例子而言，第二个CompletableFuture对象的结果取决于第一个CompletableFuture，所以无论你使用哪个版本的方法来处理CompletableFuture对象，对于最终的结果，或者大致的时间而言都没有多少差别。我们选择thenCompose方法的原因是因为它更高效一些，因为少了很多线程切换的开销。

11.4.4 将两个 CompletableFuture 对象整合起来，无论它们是否存在依赖

代码清单11-16中，你对一个CompletableFuture对象调用了thenCompose方法，并向其

传递了第二个CompletableFuture，而第二个CompletableFuture又需要使用第一个CompletableFuture的执行结果作为输入。但是，另一种比较常见的情况是，你需要将两个完全不相干的CompletableFuture对象的结果整合起来，而且你也不希望等到第一个任务完全结束才开始第二项任务。

这种情况，你应该使用thenCombine方法，它接收名为BiFunction的第二参数，这个参数定义了当两个CompletableFuture对象完成计算后，结果如何合并。同thenCompose方法一样，thenCombine方法也提供有一个Async的版本。这里，如果使用thenCombineAsync会导致BiFunction中定义的合并操作被提交到线程池中，由另一个任务以异步的方式执行。

回到我们正在运行的这个例子，你知道，有一家商店提供的价格是以欧元（EUR）计价的，但是你希望以美元的方式提供给你的客户。你可以用异步的方式向商店查询指定商品的价格，同时从远程的汇率服务那里查到欧元和美元之间的汇率。当二者都结束时，再将这两个结果结合起来，用返回的商品价格乘以当时的汇率，得到以美元计价的商品价格。用这种方式，你需要使用第三个CompletableFuture对象，当前两个CompletableFuture计算出结果，并由BiFunction方法完成合并后，由它来最终结束这一任务，代码清单如下所示。

代码清单11-17　合并两个独立的CompletableFuture对象

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD)),
            (price, rate) -> price * rate
        );
```

- 创建第一个任务查询商店取得商品的价格
- 创建第二个独立任务，查询美元和欧元之间的转换汇率
- 通过乘法整合得到的商品价格和汇率

这里整合的操作只是简单的乘法操作，用另一个单独的任务对其进行操作有些浪费资源，所以你只要使用thenCombine方法，无需特别求助于异步版本的thenCombineAsync方法。图11-6展示了代码清单11-17中创建的多个任务是如何在线程池中选择不同的线程执行的，以及它们最终的运行结果又是如何整合的。

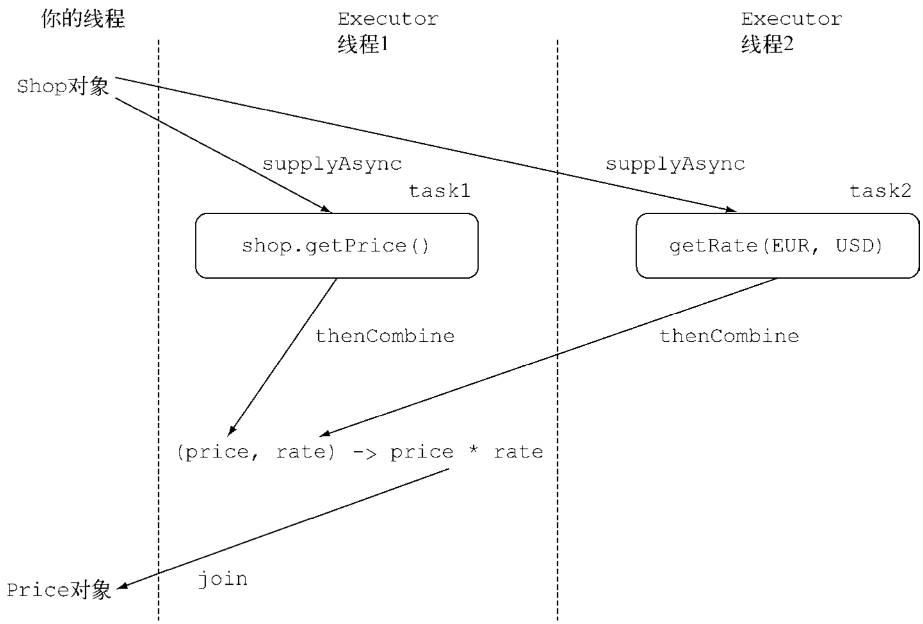


图11-6　合并两个相互独立的异步任务

11.4.5　对 **Future** 和 **CompletableFuture** 的回顾

前文介绍的最后两个例子，即代码清单11-16和代码清单11-17，非常清晰地呈现了相对于采用Java 8之前提供的Future实现，CompletableFuture版本实现所具备的巨大优势。CompletableFuture利用Lambda表达式以声明式的API提供了一种机制，能够用最有效的方式，非常容易地将多个以同步或异步方式执行复杂操作的任务结合到一起。为了更直观地感受一下使用CompletableFuture在代码可读性上带来的巨大提升，你可以尝试仅使用Java 7中提供的特性，重新实现代码清单11-17的功能。代码清单11-18展示了如何实现这一效果。

代码清单11-18　利用Java 7的方法合并两个Future对象

```
ExecutorService executor = Executors.newCachedThreadPool();
final Future<Double> futureRate = executor.submit(new Callable<Double>() {
        public Double call() {
            return exchangeService.getRate(Money.EUR, Money.USD);
        }});
Future<Double> futurePriceInUSD = executor.submit(new Callable<Double>() {
        public Double call() {
            double priceInEUR = shop.getPrice(product);
            return priceInEUR * futureRate.get();
        }});
```

- 创建一个**ExecutorService**将任务提交到线程池
- 创建一个查询欧元到美元转换汇率的**Future**
- 在第二个**Future**中查询指定商店中特定商品的价格
- 在查找价格操作的同一个**Future**中，将价格和汇率做乘法计算出汇后价格

在代码清单11-18中，你通过向执行器提交一个Callable对象的方式创建了第一个Future对象，向外部服务查询欧元和美元之间的转换汇率。紧接着，你创建了第二个Future对象，查询指定商店中特定商品的欧元价格。最终，用与代码清单11-17一样的方式，你在同一个Future中通过查询商店得到的欧元商品价格乘以汇率得到了最终的价格。注意，代码清单11-17中如果使用thenCombineAsync，不使用thenCombine，像代码清单11-18中那样，采用第三个Future单独进行商品价格和汇率的乘法运算，效果是几乎相同的。这两种实现看起来没太大区别，原因是你只对两个Future进行了合并。通过代码清单11-19和代码清单11-20，我们能看到创建流水线对同步和异步操作进行混合操作有多么简单，随着处理任务和需要合并结果数目的增加，这种声明式程序设计的优势也愈发明显。

你的“最佳价格查询器”应用基本已经完成，不过还缺失了一些元素。你会希望尽快将不同商店中的商品价格呈现给你的用户（这是车辆保险或者机票比价网站的典型需求），而不是像你之前那样，等所有的数据都完备之后再呈现。接下来的一节，你会了解如何通过响应Completable-Future的completion事件实现这一功能（与此相反，调用get或者join方法只会造成阻塞，直到CompletableFuture完成才能继续往下运行）。

11.5　响应 CompletableFuture 的 completion 事件

本章你看到的所有示例代码都是通过在响应之前添加1秒钟的等待延迟模拟方法的远程调用。毫无疑问，现实世界中，你的应用访问各个远程服务时很可能遭遇无法预知的延迟，触发的原因多种多样，从服务器的负荷到网络的延迟，有些甚至是源于远程服务如何评估你应用的商业价值，即可能相对于其他的应用，你的应用每次查询的消耗时间更长。

由于这些原因，你希望购买的商品在某些商店的查询速度要比另一些商店更快。为了说明本章的内容，我们以下面的代码清单为例，使用randomDelay方法取代原来的固定延迟。

代码清单11-19　一个模拟生成0.5秒至2.5秒随机延迟的方法

```
private static final Random random = new Random();
public static void randomDelay() {
    int delay = 500 + random.nextInt(2000);
    try {
        Thread.sleep(delay);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

目前为止，你实现的findPrices方法只有在取得所有商店的返回值时才显示商品的价格。而你希望的效果是，只要有商店返回商品价格就在第一时间显示返回值，不再等待那些还未返回的商店（有些甚至会发生超时）。你如何实现这种更进一步的改进要求呢？

11.5.1 对最佳价格查询器应用的优化

你要避免的首要问题是，等待创建一个包含了所有价格的List创建完成。你应该做的是直接处理CompletableFuture流，这样每个CompletableFuture都在为某个商店执行必要的操作。为了实现这一目标，在下面的代码清单中，你会对代码清单11-12中代码实现的第一部分进行重构，实现findPricesStream方法来生成一个由CompletableFuture构成的流。

代码清单11-20 重构findPrices方法返回一个由Future构成的流

```
public Stream<CompletableFuture<String>> findPricesStream(String product) {
    return shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(
                    () -> shop.getPrice(product), executor))
            .map(future -> future.thenApply(Quote::parse))
            .map(future -> future.thenCompose(quote ->
                    CompletableFuture.supplyAsync(
                      () -> Discount.applyDiscount(quote), executor)));
}
```

现在，你为findPricesStream方法返回的Stream添加了第四个map操作，在此之前，你已经在该方法内部调用了三次map。这个新添加的操作其实很简单，只是在每个CompletableFuture上注册一个操作，该操作会在CompletableFuture完成执行后使用它的返回值。Java 8的CompletableFuture通过thenAccept方法提供了这一功能，它接收CompletableFuture执行完毕后的返回值做参数。在这里的例子中，该值是由Discount服务返回的字符串值，它包含了提供请求商品的商店名称及折扣价格，你想要做的操作也很简单，只是将结果打印输出：

```
findPricesStream("myPhone").map(f -> f.thenAccept(System.out::println));
```

注意，和你之前看到的thenCompose和thenCombine方法一样，thenAccept方法也提供了一个异步版本，名为thenAcceptAsync。异步版本的方法会对处理结果的消费者进行调度，从线程池中选择一个新的线程继续执行，不再由同一个线程完成CompletableFuture的所有任务。因为你想要避免不必要的上下文切换，更重要的是你希望避免在等待线程上浪费时间，尽快响应CompletableFuture的completion事件，所以这里没有采用异步版本。

由于thenAccept方法已经定义了如何处理CompletableFuture返回的结果，一旦CompletableFuture计算得到结果，它就返回一个CompletableFuture<Void>。所以，map操作返回的是一个Stream<CompletableFuture<Void>>。对这个<CompletableFuture-<Void>>对象，你能做的事非常有限，只能等待其运行结束，不过这也是你所期望的。你还希望能给最慢的商店一些机会，让它有机会打印输出返回的价格。为了实现这一目的，你可以把构成Stream的所有CompletableFuture<Void>对象放到一个数组中，等待所有的任务执行完成，代码如下所示。

11

代码清单11-21　响应CompletableFuture的completion事件

```
CompletableFuture[] futures = findPricesStream("myPhone")
    .map(f -> f.thenAccept(System.out::println))
    .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
```

allOf工厂方法接收一个由CompletableFuture构成的数组，数组中的所有CompletableFuture对象执行完成之后，它返回一个CompletableFuture<Void>对象。这意味着，如果你需要等待最初Stream中的所有CompletableFuture对象执行完毕，对allOf方法返回的CompletableFuture执行join操作是个不错的主意。这个方法对“最佳价格查询器”应用也是有用的，因为你的用户可能会困惑是否后面还有一些价格没有返回，使用这个方法，你可以在执行完毕之后打印输出一条消息“All shops returned results or timed out”。

然而在另一些场景中，你可能希望只要CompletableFuture对象数组中有任何一个执行完毕就不再等待，比如，你正在查询两个汇率服务器，任何一个返回了结果都能满足你的需求。在这种情况下，你可以使用一个类似的工厂方法anyOf。该方法接收一个CompletableFuture对象构成的数组，返回由第一个执行完毕的CompletableFuture对象的返回值构成的CompletableFuture<Object>。

11.5.2　付诸实践

正如我们在本节开篇所讨论的，现在你可以通过代码清单11-19中的randomDelay方法模拟远程方法调用，产生一个介于0.5秒到2.5秒的随机延迟，不再使用恒定1秒的延迟值。代码清单11-21应用了这一改变，执行这段代码你会看到不同商店的价格不再像之前那样总是在一个时刻返回，而是随着商店折扣价格返回的顺序逐一地打印输出。为了让这一改变的效果更加明显，我们对代码进行了微调，在输出中打印每个价格计算所消耗的时间：

```
long start = System.nanoTime();
CompletableFuture[] futures = findPricesStream("myPhone27S")
    .map(f -> f.thenAccept(
        s -> System.out.println(s + " (done in " +
             ((System.nanoTime() - start) / 1_000_000) + " msecs)")))
    .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
System.out.println("All shops have now responded in "
                   + ((System.nanoTime() - start) / 1_000_000) + " msecs");
```

运行这段代码所产生的输出如下：

```
BuyItAll price is 184.74 (done in 2005 msecs)
MyFavoriteShop price is 192.72 (done in 2157 msecs)
LetsSaveBig price is 135.58 (done in 3301 msecs)
ShopEasy price is 167.28 (done in 3869 msecs)
BestPrice price is 110.93 (done in 4188 msecs)
All shops have now responded in 4188 msecs
```

我们看到，由于随机延迟的效果，第一次价格查询比最慢的查询要快两倍多。

11.6　小结

这一章中，你学到的内容如下。

- 执行比较耗时的操作时，尤其是那些依赖一个或多个远程服务的操作，使用异步任务可以改善程序的性能，加快程序的响应速度。
- 你应该尽可能地为客户提供异步API。使用`CompletableFuture`类提供的特性，你能够轻松地实现这一目标。
- `CompletableFuture`类还提供了异常管理的机制，让你有机会抛出/管理异步任务执行中发生的异常。
- 将同步API的调用封装到一个`CompletableFuture`中，你能够以异步的方式使用其结果。
- 如果异步任务之间相互独立，或者它们之间某一些的结果是另一些的输入，你可以将这些异步任务构造或者合并成一个。
- 你可以为`CompletableFuture`注册一个回调函数，在`Future`执行完毕或者它们计算的结果可用时，针对性地执行一些程序。
- 你可以决定在什么时候结束程序的运行，是等待由`CompletableFuture`对象构成的列表中所有的对象都执行完毕，还是只要其中任何一个首先完成就中止程序的运行。

第 12 章 新的日期和时间API

本章内容

- ❑ 为什么在Java 8中需要引入新的日期和时间库
- ❑ 同时为人和机器表示日期和时间
- ❑ 定义时间的度量
- ❑ 操纵、格式化以及解析日期
- ❑ 处理不同的时区和历法

Java的API提供了很多有用的组件，能帮助你构建复杂的应用。不过，Java API也不总是完美的。我们相信大多数有经验的程序员都会赞同Java 8之前的库对日期和时间的支持就非常不理想。然而，你也不用太担心：Java 8中引入全新的日期和时间API就是要解决这一问题。

在Java 1.0中，对日期和时间的支持只能依赖`java.util.Date`类。正如类名所表达的，这个类无法表示日期，只能以毫秒的精度表示时间。更糟糕的是它的易用性，由于某些原因未知的设计决策，这个类的易用性被深深地损害了，比如：年份的起始选择是1900年，月份的起始从0开始。这意味着，如果你想要用`Date`表示Java 8的发布日期，即2014年3月18日，需要创建下面这样的`Date`实例：

```
Date date = new Date(114, 2, 18);
```

它的打印输出效果为：

```
Tue Mar 18 00:00:00 CET 2014
```

看起来不那么直观，不是吗？此外，甚至`Date`类的`toString`方法返回的字符串也容易误导人。以我们的例子而言，它的返回值中甚至还包含了JVM的默认时区CET，即中欧时间（Central Europe Time）。但这并不表示`Date`类在任何方面支持时区。

随着Java 1.0退出历史舞台，`Date`类的种种问题和限制几乎一扫而光，但很明显，这些历史旧账如果不牺牲前向兼容性是无法解决的。所以，在Java 1.1中，`Date`类中的很多方法被废弃了，取而代之的是`java.util.Calendar`类。很不幸，`Calendar`类也有类似的问题和设计缺陷，导致使用这些方法写出的代码非常容易出错。比如，月份依旧是从0开始计算（不过，至少`Calendar`类拿掉了由1900年开始计算年份这一设计）。更糟的是，同时存在`Date`和`Calendar`这两个类，

也增加了程序员的困惑。到底该使用哪一个类呢？此外，有的特性只在某一个类有提供，比如用于以语言无关方式格式化和解析日期或时间的DateFormat方法就只在Date类里有。

DateFormat方法也有它自己的问题。比如，它不是线程安全的。这意味着两个线程如果尝试使用同一个formatter解析日期，你可能会得到无法预期的结果。

最后，Date和Calendar类都是可以变的。能把2014年3月18日修改成4月18日意味着什么呢？这种设计会将你拖入维护的噩梦，接下来的一章，我们会讨论函数式编程，你在该章中会了解到更多的细节。

所有这些缺陷和不一致导致用户们转投第三方的日期和时间库，比如Joda-Time。为了解决这些问题，Oracle决定在原生的Java API中提供高质量的日期和时间支持。所以，你会看到Java 8在java.time包中整合了很多Joda-Time的特性。

这一章中，我们会一起探索新的日期和时间API所提供的新特性。我们从最基本的用例入手，比如创建同时适合人与机器的日期和时间，逐渐转入到日期和时间API更高级的一些应用，比如操纵、解析、打印输出日期–时间对象，使用不同的时区和年历。

12.1 LocalDate、LocalTime、Instant、Duration 以及 Period

让我们从探索如何创建简单的日期和时间间隔入手。java.time包中提供了很多新的类可以帮你解决问题，它们是LocalDate、LocalTime、Instant、Duration和Period。

12.1.1 使用 LocalDate 和 LocalTime

开始使用新的日期和时间API时，你最先碰到的可能是LocalDate类。该类的实例是一个不可变对象，它只提供了简单的日期，并不含当天的时间信息。另外，它也不附带任何与时区相关的信息。

你可以通过静态工厂方法of创建一个LocalDate实例。LocalDate实例提供了多种方法来读取常用的值，比如年份、月份、星期几等，如下所示。

12

代码清单12-1 创建一个LocalDate对象并读取其值

```
LocalDate date = LocalDate.of(2014, 3, 18);    <— 2014-03-18
int year = date.getYear();                     <— 2014
Month month = date.getMonth();                 <— MARCH
int day = date.getDayOfMonth();                <— 18
DayOfWeek dow = date.getDayOfWeek();           <— TUESDAY
int len = date.lengthOfMonth();                <— 31 (days in March)
boolean leap = date.isLeapYear();              <— false (not a leap year)
```

你还可以使用工厂方法从系统时钟中获取当前的日期：

```
LocalDate today = LocalDate.now();
```

本章剩余的部分会探讨所有日期–时间类，这些类都提供了类似的工厂方法。你还可以通过

传递一个TemporalField参数给get方法拿到同样的信息。TemporalField是一个接口，它定义了如何访问temporal对象某个字段的值。ChronoField枚举实现了这一接口，所以你可以很方便地使用get方法得到枚举元素的值，如下所示。

代码清单12-2　使用TemporalField读取LocalDate的值

```
int year = date.get(ChronoField.YEAR);
int month = date.get(ChronoField.MONTH_OF_YEAR);
int day = date.get(ChronoField.DAY_OF_MONTH);
```

类似地，一天中的时间，比如13:45:20，可以使用LocalTime类表示。你可以使用of重载的两个工厂方法创建LocalTime的实例。第一个重载函数接收小时和分钟，第二个重载函数同时还接收秒。同LocalDate一样，LocalTime类也提供了一些getter方法访问这些变量的值，如下所示。

代码清单12-3　创建LocalTime并读取其值

```
LocalTime time = LocalTime.of(13, 45, 20);   <— 13:45:20
int hour = time.getHour();                 <— 13
int minute = time.getMinute();          <— 45
int second = time.getSecond();      <— 20
```

LocalDate和LocalTime都可以通过解析代表它们的字符串创建。使用静态方法parse，你可以实现这一目的：

```
LocalDate date = LocalDate.parse("2014-03-18");
LocalTime time = LocalTime.parse("13:45:20");
```

你可以向parse方法传递一个DateTimeFormatter。该类的实例定义了如何格式化一个日期或者时间对象。正如我们之前所介绍的，它是替换老版java.util.DateFormat的推荐替代品。我们会在12.2节展开介绍怎样使用DateTimeFormatter。同时，也请注意，一旦传递的字符串参数无法被解析为合法的LocalDate或LocalTime对象，这两个parse方法都会抛出一个继承自RuntimeException的DateTimeParseException异常。

12.1.2　合并日期和时间

这个复合类名叫LocalDateTime，是LocalDate和LocalTime的合体。它同时表示了日期和时间，但不带有时区信息，你可以直接创建，也可以通过合并日期和时间对象构造，如下所示。

代码清单12-4　直接创建LocalDateTime对象，或者通过合并日期和时间的方式创建

```
// 2014-03-18T13:45:20
LocalDateTime dt1 = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45, 20);
LocalDateTime dt2 = LocalDateTime.of(date, time);
LocalDateTime dt3 = date.atTime(13, 45, 20);
LocalDateTime dt4 = date.atTime(time);
LocalDateTime dt5 = time.atDate(date);
```

注意，通过它们各自的atTime或者atDate方法，向LocalDate传递一个时间对象，或者向

LocalTime传递一个日期对象的方式，你可以创建一个LocalDateTime对象。你也可以使用toLocalDate或者toLocalTime方法，从LocalDateTime中提取LocalDate或者LocalTime组件：

```
LocalDate date1 = dt1.toLocalDate();        <— 2014-03-18
LocalTime time1 = dt1.toLocalTime();        <— 13:45:20
```

12.1.3 机器的日期和时间格式

作为人，我们习惯于以星期几、几号、几点、几分这样的方式理解日期和时间。毫无疑问，这种方式对于计算机而言并不容易理解。从计算机的角度来看，建模时间最自然的格式是表示一个持续时间段上某个点的单一大整型数。这也是新的java.time.Instant类对时间建模的方式，基本上它是以Unix元年时间（传统的设定为UTC时区1970年1月1日午夜时分）开始所经历的秒数进行计算。

你可以通过向静态工厂方法ofEpochSecond传递一个代表秒数的值创建一个该类的实例。静态工厂方法ofEpochSecond还有一个增强的重载版本，它接收第二个以纳秒为单位的参数值，对传入作为秒数的参数进行调整。重载的版本会调整纳秒参数，确保保存的纳秒分片在0到999 999 999之间。这意味着下面这些对ofEpochSecond工厂方法的调用会返回几乎同样的Instant对象：

```
Instant.ofEpochSecond(3);
Instant.ofEpochSecond(3, 0);
Instant.ofEpochSecond(2, 1_000_000_000);     <— 2秒之后再加上100万纳秒（1秒）
Instant.ofEpochSecond(4, -1_000_000_000);    <— 4秒之前的100万纳秒（1秒）
```

正如你已经在LocalDate及其他为便于阅读而设计的日期–时间类中所看到的那样，Instant类也支持静态工厂方法now，它能够帮你获取当前时刻的时间戳。我们想要特别强调一点，Instant的设计初衷是为了便于机器使用。它包含的是由秒及纳秒所构成的数字。所以，它无法处理那些我们非常容易理解的时间单位。比如下面这段语句：

```
int day = Instant.now().get(ChronoField.DAY_OF_MONTH);
```

它会抛出下面这样的异常：

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field:
    DayOfMonth
```

但是你可以通过Duration和Period类使用Instant，接下来我们会对这部分内容进行介绍。

12.1.4 定义 Duration 或 Period

目前为止，你看到的所有类都实现了Temporal接口，Temporal接口定义了如何读取和操纵为时间建模的对象的值。之前的介绍中，我们已经了解了创建Temporal实例的几种方法。很自然地你会想到，我们需要创建两个Temporal对象之间的duration。Duration类的静态工厂方法between就是为这个目的而设计的。你可以创建两个LocalTimes对象、两个LocalDateTimes

对象，或者两个Instant对象之间的duration，如下所示：

```
Duration d1 = Duration.between(time1, time2);
Duration d1 = Duration.between(dateTime1, dateTime2);
Duration d2 = Duration.between(instant1, instant2);
```

由于LocalDateTime和Instant是为不同的目的而设计的，一个是为了便于人阅读使用，另一个是为了便于机器处理，所以你不能将二者混用。如果你试图在这两类对象之间创建duration，会触发一个DateTimeException异常。此外，由于Duration类主要用于以秒和纳秒衡量时间的长短，你不能仅向between方法传递一个LocalDate对象做参数。

如果你需要以年、月或者日的方式对多个时间单位建模，可以使用Period类。使用该类的工厂方法between，你可以使用得到两个LocalDate之间的时长，如下所示：

```
Period tenDays = Period.between(LocalDate.of(2014, 3, 8),
                                LocalDate.of(2014, 3, 18));
```

最后，Duration和Period类都提供了很多非常方便的工厂类，直接创建对应的实例；换句话说，就像下面这段代码那样，不再是只能以两个temporal对象的差值的方式来定义它们的对象。

代码清单12-5　创建Duration和Period对象

```
Duration threeMinutes = Duration.ofMinutes(3);
Duration threeMinutes = Duration.of(3, ChronoUnit.MINUTES);

Period tenDays = Period.ofDays(10);
Period threeWeeks = Period.ofWeeks(3);
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

Duration类和Period类共享了很多相似的方法，参见表12-1所示。

表12-1　日期–时间类中表示时间间隔的通用方法

方　法　名	是否是静态方法	方法描述
between	是	创建两个时间点之间的 interval
from	是	由一个临时时间点创建 interval
of	是	由它的组成部分创建 interval 的实例
parse	是	由字符串创建 interval 的实例
addTo	否	创建该 interval 的副本，并将其叠加到某个指定的 temporal 对象
get	否	读取该 interval 的状态
isNegative	否	检查该 interval 是否为负值，不包含零
isZero	否	检查该 interval 的时长是否为零
minus	否	通过减去一定的时间创建该 interval 的副本
multipliedBy	否	将 interval 的值乘以某个标量创建该 interval 的副本
negated	否	以忽略某个时长的方式创建该 interval 的副本

（续）

方 法 名	是否是静态方法	方法描述
plus	否	以增加某个指定的时长的方式创建该 interval 的副本
subtractFrom	否	从指定的 temporal 对象中减去该 interval

截至目前，我们介绍的这些日期–时间对象都是不可修改的，这是为了更好地支持函数式编程，确保线程安全，保持领域模式一致性而做出的重大设计决定。当然，新的日期和时间API也提供了一些便利的方法来创建这些对象的可变版本。比如，你可能希望在已有的LocalDate实例上增加3天。我们在下一节中会针对这一主题进行介绍。除此之外，我们还会介绍如何依据指定的模式，比如dd/MM/yyyy，创建日期–时间格式器，以及如何使用这种格式器解析和输出日期。

12.2 操纵、解析和格式化日期

如果你已经有一个LocalDate对象，想要创建它的一个修改版，最直接也最简单的方法是使用withAttribute方法。withAttribute方法会创建对象的一个副本，并按照需要修改它的属性。注意，下面的这段代码中所有的方法都返回一个修改了属性的对象。它们都不会修改原来的对象！

代码清单12-6 以比较直观的方式操纵LocalDate的属性

```
LocalDate date1 = LocalDate.of(2014, 3, 18);           <— 2014-03-18
LocalDate date2 = date1.withYear(2011);                <— 2011-03-18
LocalDate date3 = date2.withDayOfMonth(25);            <— 2011-03-25
LocalDate date4 = date3.with(ChronoField.MONTH_OF_YEAR, 9);   <— 2011-09-25
```

采用更通用的with方法能达到同样的目的，它接受的第一个参数是一个TemporalField对象，格式类似代码清单12-6的最后一行。最后这一行中使用的with方法和代码清单12-2中的get方法有些类似。它们都声明于Temporal接口，所有的日期和时间API类都实现这两个方法，它们定义了单点的时间，比如LocalDate、LocalTime、LocalDateTime以及Instant。更确切地说，使用get和with方法，我们可以将Temporal对象值的读取和修改区分开。如果Temporal对象不支持请求访问的字段，它会抛出一个UnsupportedTemporalTypeException异常，比如试图访问Instant对象的ChronoField.MONTH_OF_YEAR字段，或者LocalDate对象的ChronoField.NANO_OF_SECOND字段时都会抛出这样的异常。

它甚至能以声明的方式操纵LocalDate对象。比如，你可以像下面这段代码那样加上或者减去一段时间。

代码清单12-7 以相对方式修改LocalDate对象的属性

```
LocalDate date1 = LocalDate.of(2014, 3, 18);           <— 2014-03-18
LocalDate date2 = date1.plusWeeks(1);                  <— 2014-03-25
LocalDate date3 = date2.minusYears(3);                 <— 2011-03-25
LocalDate date4 = date3.plus(6, ChronoUnit.MONTHS);    <— 2011-09-25
```

12

与我们刚才介绍的get和with方法类似，代码清单12-7中最后一行使用的plus方法也是通用方法，它和minus方法都声明于Temporal接口中。通过这些方法，对TemporalUnit对象加上或者减去一个数字，我们能非常方便地将Temporal对象前溯或者回滚至某个时间段，通过ChronoUnit枚举我们可以非常方便地实现TemporalUnit接口。

大概你已经猜到，像LocalDate、LocalTime、LocalDateTime以及Instant这样表示时间点的日期–时间类提供了大量通用的方法，表12-2对这些通用的方法进行了总结。

表12-2　表示时间点的日期–时间类的通用方法

方　法　名	是否是静态方法	描　　述
from	是	依据传入的 Temporal 对象创建对象实例
now	是	依据系统时钟创建 Temporal 对象
of	是	由 Temporal 对象的某个部分创建该对象的实例
parse	是	由字符串创建 Temporal 对象的实例
atOffset	否	将 Temporal 对象和某个时区偏移相结合
atZone	否	将 Temporal 对象和某个时区相结合
format	否	使用某个指定的格式器将 Temporal 对象转换为字符串（Instant 类不提供该方法）
get	否	读取 Temporal 对象的某一部分的值
minus	否	创建 Temporal 对象的一个副本，通过将当前 Temporal 对象的值减去一定的时长创建该副本
plus	否	创建 Temporal 对象的一个副本，通过将当前 Temporal 对象的值加上一定的时长创建该副本
with	否	以该 Temporal 对象为模板，对某些状态进行修改创建该对象的副本

你可以尝试一下测验12.1，检查一下到目前为止你都掌握了哪些操纵日期的技能。

测验12.1　操纵LocalDate对象

经过下面这些操作，date变量的值是什么？

```
LocalDate date = LocalDate.of(2014, 3, 18);
date = date.with(ChronoField.MONTH_OF_YEAR, 9);
date = date.plusYears(2).minusDays(10);
date.withYear(2011);
```

答案：2016-09-08。

正如我们刚才看到的，你可以通过绝对的方式，也能以相对的方式操纵日期。你甚至还可以在一个语句中连接多个操作，因为每个动作都会创建一个新的LocalDate对象，后续的方法调用可以操纵前一方法创建的对象。这段代码的最后一句不会产生任何我们能看到的效果，因为它像前面的那些操作一样，会创建一个新的LocalDate实例，不过我们并没有将这个新创建的值赋给任何的变量。

12.2.1 使用 TemporalAdjuster

截至目前，你所看到的所有日期操作都是相对比较直接的。有的时候，你需要进行一些更加复杂的操作，比如，将日期调整到下个周日、下个工作日，或者是本月的最后一天。这时，你可以使用重载版本的with方法，向其传递一个提供了更多定制化选择的TemporalAdjuster对象，更加灵活地处理日期。对于最常见的用例，日期和时间API已经提供了大量预定义的TemporalAdjuster。你可以通过TemporalAdjuster类的静态工厂方法访问它们，如下所示。

代码清单12-8 使用预定义的TemporalAdjuster

```
import static java.time.temporal.TemporalAdjusters.*;

LocalDate date1 = LocalDate.of(2014, 3, 18);                      <-- 2014-03-18
LocalDate date2 = date1.with(nextOrSame(DayOfWeek.SUNDAY));  <-- 2014-03-23
LocalDate date3 = date2.with(lastDayOfMonth());              <-- 2014-03-31
```

表12-3提供了TemporalAdjuster中包含的工厂方法列表。

表12-3 TemporalAdjuster类中的工厂方法

方 法 名	描 述
dayOfWeekInMonth	创建一个新的日期，它的值为同一个月中每一周的第几天
firstDayOfMonth	创建一个新的日期，它的值为当月的第一天
firstDayOfNextMonth	创建一个新的日期，它的值为下月的第一天
firstDayOfNextYear	创建一个新的日期，它的值为明年的第一天
firstDayOfYear	创建一个新的日期，它的值为当年的第一天
firstInMonth	创建一个新的日期，它的值为同一个月中，第一个符合星期几要求的值
lastDayOfMonth	创建一个新的日期，它的值为当月的最后一天
lastDayOfNextMonth	创建一个新的日期，它的值为下月的最后一天
lastDayOfNextYear	创建一个新的日期，它的值为明年的最后一天
lastDayOfYear	创建一个新的日期，它的值为今年的最后一天
lastInMonth	创建一个新的日期，它的值为同一个月中，最后一个符合星期几要求的值
next/previous	创建一个新的日期，并将其值设定为日期调整后或者调整前，第一个符合指定星期几要求的日期
nextOrSame/previousOrSame	创建一个新的日期，并将其值设定为日期调整后或者调整前，第一个符合指定星期几要求的日期，如果该日期已经符合要求，直接返回该对象

正如我们看到的，使用TemporalAdjuster我们可以进行更加复杂的日期操作，而且这些方法的名称也非常直观，方法名基本就是问题陈述。此外，即使你没有找到符合你要求的预定义的TemporalAdjuster，创建你自己的TemporalAdjuster也并非难事。实际上，Temporal-Adjuster接口只声明了单一的一个方法（这使得它成为了一个函数式接口），定义如下。

代码清单12-9　TemporalAdjuster接口

```
@FunctionalInterface
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

这意味着TemporalAdjuster接口的实现需要定义如何将一个Temporal对象转换为另一个Temporal对象。你可以把它看成一个UnaryOperator<Temporal>。花几分钟时间完成测验12.2，练习一下我们到目前为止所学习的东西，请实现你自己的TemporalAdjuster。

测验12.2　实现一个定制的TemporalAdjuster

请设计一个NextWorkingDay类，该类实现了TemporalAdjuster接口，能够计算明天的日期，同时过滤掉周六和周日这些节假日。格式如下所示：

```
date = date.with(new NextWorkingDay());
```

如果当天的星期介于周一至周五之间，日期向后移动一天；如果当天是周六或者周日，则返回下一个周一。

答案：下面是参考的NextWorkingDay类的实现。

```
public class NextWorkingDay implements TemporalAdjuster {
    @Override
    public Temporal adjustInto(Temporal temporal) {
        DayOfWeek dow =
                DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));   <- 读取当前日期
        int dayToAdd = 1;                                              <- 正常情况，增加1天
        if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;                     <- 如果当天是周五，增加3天
        else if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;              <- 如果当天是周六，增加2天
        return temporal.plus(dayToAdd, ChronoUnit.DAYS);               <- 增加恰当的天数后，返回修改的日期
    }
}
```

该TemporalAdjuster通常情况下将日期往后顺延一天，如果当天是周六或者周日，则依据情况分别将日期顺延3天或者2天。注意，由于TemporalAdjuster是一个函数式接口，你只能以Lambda表达式的方式向该adjuster接口传递行为：

```
date = date.with(temporal -> {
        DayOfWeek dow =
                DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
    int dayToAdd = 1;
    if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;
    else if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;
    return temporal.plus(dayToAdd, ChronoUnit.DAYS);
});
```

你大概会希望在你代码的多个地方使用同样的方式去操作日期，为了达到这一目的，我们建议你像我们的示例那样将它的逻辑封装到一个类中。对于你经常使用的操作，都应该采用类似的方式，进行封装。最终，你会创建自己的类库，让你和你的团队能轻松地实现代码复用。

如果你想要使用Lambda表达式定义TemporalAdjuster对象，推荐使用Temporal-Adjusters类的静态工厂方法ofDateAdjuster，它接受一个UnaryOperator<LocalDate>

类型的参数，代码如下：

```
TemporalAdjuster nextWorkingDay = TemporalAdjusters.ofDateAdjuster(
    temporal -> {
        DayOfWeek dow =
            DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
        int dayToAdd = 1;
        if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;
        if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;
        return temporal.plus(dayToAdd, ChronoUnit.DAYS);
    });

date = date.with(nextWorkingDay);
```

你可能希望对你的日期时间对象进行的另外一个通用操作是，依据你的业务领域以不同的格式打印输出这些日期和时间对象。类似地，你可能也需要将那些格式的字符串转换为实际的日期对象。接下来的一节，我们会演示新的日期和时间API提供那些机制是如何完成这些任务的。

12.2.2 打印输出及解析日期–时间对象

处理日期和时间对象时，格式化以及解析日期–时间对象是另一个非常重要的功能。新的java.time.format包就是特别为这个目的而设计的。这个包中，最重要的类是DateTimeFormatter。创建格式器最简单的方法是通过它的静态工厂方法以及常量。像BASIC_ISO_DATE和ISO_LOCAL_DATE这样的常量是DateTimeFormatter类的预定义实例。所有的DateTimeFormatter实例都能用于以一定的格式创建代表特定日期或时间的字符串。比如，下面的这个例子中，我们使用了两个不同的格式器生成了字符串：

```
LocalDate date = LocalDate.of(2014, 3, 18);
String s1 = date.format(DateTimeFormatter.BASIC_ISO_DATE);  <-- 20140318
String s2 = date.format(DateTimeFormatter.ISO_LOCAL_DATE);  <-- 2014-03-18
```

你也可以通过解析代表日期或时间的字符串重新创建该日期对象。所有的日期和时间API都提供了表示时间点或者时间段的工厂方法，你可以使用工厂方法parse达到重创该日期对象的目的：

```
LocalDate date1 = LocalDate.parse("20140318",
                                 DateTimeFormatter.BASIC_ISO_DATE);
LocalDate date2 = LocalDate.parse("2014-03-18",
                                 DateTimeFormatter.ISO_LOCAL_DATE);
```

和老的java.util.DateFormat相比较，所有的DateTimeFormatter实例都是线程安全的。所以，你能够以单例模式创建格式器实例，就像DateTimeFormatter所定义的那些常量，并能在多个线程间共享这些实例。DateTimeFormatter类还支持一个静态工厂方法，它可以按照某个特定的模式创建格式器，代码清单如下。

代码清单12-10 按照某个模式创建DateTimeFormatter

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

```
LocalDate date1 = LocalDate.of(2014, 3, 18);
String formattedDate = date1.format(formatter);
LocalDate date2 = LocalDate.parse(formattedDate, formatter);
```

这段代码中，LocalDate的formate方法使用指定的模式生成了一个代表该日期的字符串。紧接着，静态的parse方法使用同样的格式器解析了刚才生成的字符串，并重建了该日期对象。ofPattern方法也提供了一个重载的版本，使用它你可以创建某个Locale的格式器，代码清单如下所示。

代码清单12-11　创建一个本地化的DateTimeFormatter

```
DateTimeFormatter italianFormatter =
               DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.ITALIAN);
LocalDate date1 = LocalDate.of(2014, 3, 18);
String formattedDate = date.format(italianFormatter); // 18. marzo 2014
LocalDate date2 = LocalDate.parse(formattedDate, italianFormatter);
```

最后，如果你还需要更加细粒度的控制，DateTimeFormatterBuilder类还提供了更复杂的格式器，你可以选择恰当的方法，一步一步地构造自己的格式器。另外，它还提供了非常强大的解析功能，比如区分大小写的解析、柔性解析（允许解析器使用启发式的机制去解析输入，不精确地匹配指定的模式）、填充，以及在格式器中指定可选节。比如，你可以通过DateTimeFormatterBuilder自己编程实现我们在代码清单12-11中使用的italianFormatter，代码清单如下。

代码清单12-12　构造一个DateTimeFormatter

```
DateTimeFormatter italianFormatter = new DateTimeFormatterBuilder()
        .appendText(ChronoField.DAY_OF_MONTH)
        .appendLiteral(". ")
        .appendText(ChronoField.MONTH_OF_YEAR)
        .appendLiteral(" ")
        .appendText(ChronoField.YEAR)
        .parseCaseInsensitive()
        .toFormatter(Locale.ITALIAN);
```

目前为止，你已经学习了如何创建、操纵、格式化以及解析时间点和时间段，但是你还不了解如何处理日期和时间之间的微妙关系。比如，你可能需要处理不同的时区，或者由于不同的历法系统带来的差异。接下来的一节，我们会探究如何使用新的日期和时间API解决这些问题。

12.3　处理不同的时区和历法

之前你看到的日期和时间的种类都不包含时区信息。时区的处理是新版日期和时间API新增加的重要功能，使用新版日期和时间API时区的处理被极大地简化了。新的java.time.ZoneId类是老版java.util.TimeZone的替代品。它的设计目标就是要让你无需为时区处理的复杂和繁琐而操心，比如处理日光时（Daylight Saving Time，DST）这种问题。跟其他日期和时间类一样，ZoneId类也是无法修改的。

时区是按照一定的规则将区域划分成的标准时间相同的区间。在ZoneRules这个类中包含了40个这样的实例。你可以简单地通过调用ZoneId的getRules()得到指定时区的规则。每个特定的ZoneId对象都由一个地区ID标识，比如：

```
ZoneId romeZone = ZoneId.of("Europe/Rome");
```

地区ID都为“{区域}/{城市}”的格式，这些地区集合的设定都由英特网编号分配机构（IANA）的时区数据库提供。你可以通过Java 8的新方法toZoneId将一个老的时区对象转换为ZoneId：

```
ZoneId zoneId = TimeZone.getDefault().toZoneId();
```

一旦得到一个ZoneId对象，你就可以将它与LocalDate、LocalDateTime或者是Instant对象整合起来，构造为一个ZonedDateTime实例，它代表了相对于指定时区的时间点，代码清单如下所示。

代码清单12-13 为时间点添加时区信息

```
LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
ZonedDateTime zdt1 = date.atStartOfDay(romeZone);

LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
ZonedDateTime zdt2 = dateTime.atZone(romeZone);

Instant instant = Instant.now();
ZonedDateTime zdt3 = instant.atZone(romeZone);
```

图12-1对ZonedDateTime的组成部分进行了说明，相信能够帮助你理解LocaleDate、LocalTime、LocalDateTime以及ZoneId之间的差异。

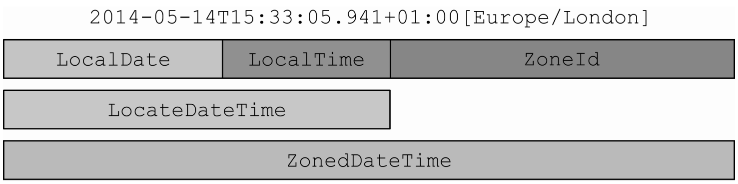


图12-1 理解ZonedDateTime

通过ZoneId，你还可以将LocalDateTime转换为Instant：

```
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
Instant instantFromDateTime = dateTime.toInstant(romeZone);
```

你也可以通过反向的方式得到LocalDateTime对象：

```
Instant instant = Instant.now();
LocalDateTime timeFromInstant = LocalDateTime.ofInstant(instant, romeZone);
```

12.3.1 利用和UTC/格林尼治时间的固定偏差计算时区

另一种比较通用的表达时区的方式是利用当前时区和UTC/格林尼治的固定偏差。比如，基

于这个理论，你可以说“纽约落后于伦敦5小时”。这种情况下，你可以使用ZoneOffset类，它是ZoneId的一个子类，表示的是当前时间和伦敦格林尼治子午线时间的差异：

```
ZoneOffset newYorkOffset = ZoneOffset.of("-05:00");
```

“–05:00”的偏差实际上对应的是美国东部标准时间。注意，使用这种方式定义的ZoneOffset并未考虑任何日光时的影响，所以在大多数情况下，不推荐使用。由于ZoneOffset也是ZoneId，所以你可以像代码清单12-13那样使用它。你甚至还可以创建这样的OffsetDateTime，它使用ISO-8601的历法系统，以相对于UTC/格林尼治时间的偏差方式表示日期时间。

```
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
OffsetDateTime dateTimeInNewYork = OffsetDateTime.of(date, newYorkOffset);
```

新版的日期和时间API还提供了另一个高级特性，即对非ISO历法系统（non-ISO calendaring）的支持。

12.3.2　使用别的日历系统

ISO-8601日历系统是世界文明日历系统的事实标准。但是，Java 8中另外还提供了4种其他的日历系统。这些日历系统中的每一个都有一个对应的日志类，分别是ThaiBuddhistDate、MinguoDate、JapaneseDate以及HijrahDate。所有这些类以及LocalDate都实现了ChronoLocalDate接口，能够对公历的日期进行建模。利用LocalDate对象，你可以创建这些类的实例。更通用地说，使用它们提供的静态工厂方法，你可以创建任何一个Temporal对象的实例，如下所示：

```
LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
JapaneseDate japaneseDate = JapaneseDate.from(date);
```

或者，你还可以为某个Locale显式地创建日历系统，接着创建该Locale对应的日期的实例。新的日期和时间API中，Chronology接口建模了一个日历系统，使用它的静态工厂方法ofLocale，可以得到它的一个实例，代码如下：

```
Chronology japaneseChronology = Chronology.ofLocale(Locale.JAPAN);
ChronoLocalDate now = japaneseChronology.dateNow();
```

日期及时间API的设计者建议我们使用LocalDate，尽量避免使用ChronoLocalDate，原因是开发者在他们的代码中可能会做一些假设，而这些假设在不同的日历系统中，有可能不成立。比如，有人可能会做这样的假设，即一个月天数不会超过31天，一年包括12个月，或者一年中包含的月份数目是固定的。由于这些原因，我们建议你尽量在你的应用中使用LocalDate，包括存储、操作、业务规则的解读；不过如果你需要将程序的输入或者输出本地化，这时你应该使用ChronoLocalDate类。

伊斯兰教日历

在Java 8新添加的几种日历类型中，HijrahDate（伊斯兰教日历）是最复杂一个，因为它会发生各种变化。Hijrah日历系统构建于农历月份继承之上。Java 8提供了多种方法判断一个月

份，比如新月，在世界的哪些地方可见，或者说它只能首先可见于沙特阿拉伯。withVariant方法可以用于选择期望的变化。为了支持HijrahDate这一标准，Java 8中还包括了乌姆库拉（Umm Al-Qura）变量。

下面这段代码作为一个例子说明了如何在ISO日历中计算当前伊斯兰年中斋月的起始和终止日期：

```
HijrahDate ramadanDate =
    HijrahDate.now().with(ChronoField.DAY_OF_MONTH, 1)
                    .with(ChronoField.MONTH_OF_YEAR, 9);   <-- 取得当前的Hijrah日期，紧接着对其进行修正，得到斋月的第一天，即第9个月

System.out.println("Ramadan starts on " +
                   IsoChronology.INSTANCE.date(ramadanDate) +
                   " and ends on " +
                   IsoChronology.INSTANCE.date(
                       ramadanDate.with(
                           TemporalAdjusters.lastDayOfMonth())));
```

IsoChronology.INSTANCE是IsoChronology类的一个静态实例

斋月始于2014-06-28，止于2014-07-27

12.4 小结

这一章中，你应该掌握下面这些内容。

- Java 8之前老版的java.util.Date类以及其他用于建模日期时间的类有很多不一致及设计上的缺陷，包括易变性以及糟糕的偏移值、默认值和命名。
- 新版的日期和时间API中，日期–时间对象是不可变的。
- 新的API提供了两种不同的时间表示方式，有效地区分了运行时人和机器的不同需求。
- 你可以用绝对或者相对的方式操纵日期和时间，操作的结果总是返回一个新的实例，老的日期时间对象不会发生变化。
- TemporalAdjuster让你能够用更精细的方式操纵日期，不再局限于一次只能改变它的一个值，并且你还可按照需求定义自己的日期转换器。
- 你现在可以按照特定的格式需求，定义自己的格式器，打印输出或者解析日期–时间对象。这些格式器可以通过模板创建，也可以自己编程创建，并且它们都是线程安全的。
- 你可以用相对于某个地区/位置的方式，或者以与UTC/格林尼治时间的绝对偏差的方式表示时区，并将其应用到日期–时间对象上，对其进行本地化。
- 你现在可以使用不同于ISO-8601标准系统的其他日历系统了。

Part 4

第四部分

超越 Java 8

在本书的最后一部分，我们简单地介绍Java中的函数式编程，并对Java 8和Scala中相关的特性进行比较。

第13章中，我们会全面地介绍函数式编程，介绍它的术语，并详细介绍如何在Java 8中进行函数式编程。

第14章会讨论函数式编程的一些高级技术，包括高阶函数、科里化、持久化数据结构、延迟列表，以及模式匹配。你可以将这一章看作一道混合大餐，它既包含了能直接应用到你代码中的实战技巧，也囊括了一些学术性的知识，帮助你成为知识更加渊博的程序员。

第15章讨论Java 8和Scala语言的特性比较——Scala是一种新型语言，它和Java有几分相似，都构建于JVM之上，最近一段时间发展很迅猛，在编程生态系统中已经对Java某些方面的固有地位造成了威胁。

最后，我们在第16章回顾了学习Java 8的旅程，以及向函数式编程转变的潮流。除此之外，我们还展望了会有哪些改进以及重要的新的特性可能出现在Java 8之后的版本里。

第 13 章

函数式的思考

13

本章内容

- 为什么要进行函数式编程
- 什么是函数式编程
- 声明式编程以及引用透明性
- 编写函数式Java的准则
- 迭代和递归

你已经发现了，本书中频繁地出现“函数式”这个术语。到目前为止，你可能也对函数式编程包含哪些内容有了一定的了解。它指的是Lambda表达式和一等函数吗？还是说限制你对可变对象的修改？如果是这样，采用函数式编程能为你带来什么好处呢？这一章中，我们会一一为你解答这些问题。我们会介绍什么是函数式编程，以及它的一些术语。我们首先会探究函数式编程背后的概念，比如副作用、不变性、声明式编程、引用透明性，并将它们和Java 8的实践相结合。下一章，我们会更深入地研究函数式编程的技术，包括高阶函数、科里化、持久化数据结构、延迟列表、模式匹配以及结合器。

13.1 实现和维护系统

让我们假设你被要求对一个大型的遗留软件系统进行升级，而且这个系统你之前并不是非常了解。你是否应该接受维护这种软件系统的工作呢？稍有理智的外包Java程序员只会依赖如下这种言不由衷的格言做决定，“搜索一下代码中有没有使用`synchronized`关键字，如果有就直接拒绝（由此我们可以了解修复并发导致的缺陷有多困难），否则进一步看看系统结构的复杂程度”。我们会在下面中提供更多的细节，但是你发现了吗，正如我们在前面几章所讨论的，如果你喜欢无状态的行为（即你处理`Stream`的流水线中的函数不会由于需要等待从另一个方法中读取变量，或者由于需要写入的变量同时有另一个方法正在写而发生中断），Java 8中新增的`Stream`提供了强大的技术支撑，让我们无需担心锁引起的各种问题，充分发掘系统的并发能力。

为了让程序易于使用，你还希望它具备哪些特性呢？你会希望它具有良好的结构，最好类的结构应该反映出系统的结构，这样能便于大家理解；甚至软件工程中还提供了指标，对结构的合

理性进行评估，比如耦合性（软件系统中各组件之间是否相互独立）以及内聚性（系统的各相关部分之间如何协作）。

不过，对大多数程序员而言，最关心的日常要务是代码维护时的调试：代码遭遇一些无法预期的值就有可能发生崩溃。为什么会发生这种情况？它是如何进入到这种状态的？想想看你有多少代码维护的顾虑都能归咎到这一类！[①]很明显，函数式编程提出的“无副作用”以及“不变性”对于解决这一难题是大有裨益的。让我们就此展开进一步的探讨。

13.1.1　共享的可变数据

最终，我们刚才讨论的无法预知的变量修改问题，都源于共享的数据结构被你所维护的代码中的多个方法读取和更新。假设几个类同时都保存了指向某个列表的引用。那么到底谁对这个列表拥有所属权呢？如果一个类对它进行了修改，会发生什么情况？其他的类预期会发生这种变化吗？其他的类又如何得知列表发生了修改呢？我们需要通知使用该列表的所有类这一变化吗？抑或是不是每个类都应该为自己准备一份防御式的数据备份以备不时之需呢？换句话说，由于使用了可变的共享数据结构，我们很难追踪你程序的各个组成部分所发生的变化。图13-1解释了这一问题。

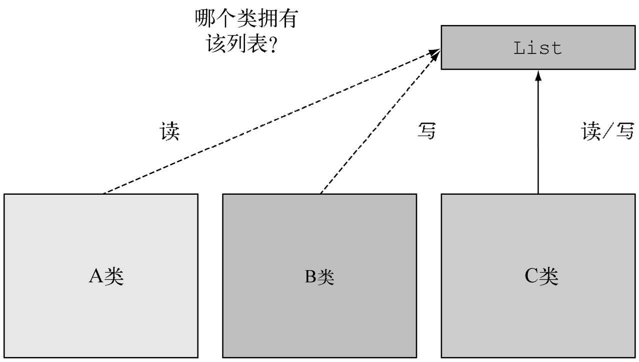


图13-1　多个类同时共享的一个可变对象。我们很难说到底哪个类真正拥有该对象

假设有这样一个系统，它不修改任何数据。维护这样的一个系统将是一个无以伦比的美梦，因为你不再会收到任何由于某些对象在某些地方修改了某个数据结构而导致的意外报告。如果一个方法既不修改它内嵌类的状态，也不修改其他对象的状态，使用`return`返回所有的计算结果，那么我们称其为纯粹的或者无副作用的。

更确切地讲，到底哪些因素会造成副作用呢？简而言之，副作用就是函数的效果已经超出了函数自身的范畴。下面是一些例子。

- 除了构造器内的初始化操作，对类中数据结构的任何修改，包括字段的赋值操作（一个典型的例子是`setter`方法）。

13

① 推荐你阅读Michael Feathers的*Working Effectively with Legacy Code*详细了解这个话题。

- 抛出一个异常。
- 进行输入/输出操作，比如向一个文件写数据。

从另一个角度来看"无副作用"的话，我们就应该考虑不可变对象。不可变对象是这样一种对象，它们一旦完成初始化就不会被任何方法修改状态。这意味着一旦一个不可变对象初始化完毕，它永远不会进入到一个无法预期的状态。你可以放心地共享它，无需保留任何副本，并且由于它们不会被修改，还是线程安全的。

"无副作用"这个想法的限制看起来很严苛，你甚至可能会质疑是否有真正的生产系统能够以这种方式构建。我们希望结束本章的学习之后，你能够确信这一点。一个好消息是，如果构成系统的各个组件都能遵守这一原则，该系统就能在完全无锁的情况下，使用多核的并发机制，因为任何一个方法都不会对其他的方法造成干扰。此外，这还是一个让你了解你的程序中哪些部分是相互独立的非常棒的机会。

这些思想都源于函数式编程，我们在下一节会进行介绍。但是在开始之前，让我们先看看函数式编程的基石*声明式编程*吧。

13.1.2　声明式编程

一般通过编程实现一个系统，有两种思考方式。一种专注于如何实现，比如："首先做这个，紧接着更新那个，然后……"举个例子，如果你希望通过计算找出列表中最昂贵的事务，通常需要执行一系列的命令：从列表中取出一个事务，将其与临时最昂贵事务进行比较；如果该事务开销更大，就将临时最昂贵的事务设置为该事务；接着从列表中取出下一个事务，并重复上述操作。

这种"如何做"风格的编程非常适合经典的面向对象编程，有些时候我们也称之为"命令式"编程，因为它的特点是它的指令和计算机底层的词汇非常相近，比如赋值、条件分支以及循环，就像下面这段代码：

```
Transaction mostExpensive = transactions.get(0);
if(mostExpensive == null)
    throw new IllegalArgumentException("Empty list of transactions")

for(Transaction t: transactions.subList(1, transactions.size())){
    if(t.getValue() > mostExpensive.getValue()){
        mostExpensive = t;
    }
}
```

另一种方式则更加关注要做什么。你在第4章和第5章中已经看到，使用Stream API你可以指定下面这样的查询：

```
Optional<Transaction> mostExpensive =
    transactions.stream()
                .max(comparing(Transaction::getValue));
```

这个查询把最终如何实现的细节留给了函数库。我们把这种思想称之为*内部迭代*。它的巨大优势在于你的查询语句现在读起来就像是问题陈述，由于采用了这种方式，我们马上就能理解它

的功能，比理解一系列的命令要简洁得多。

采用这种“要做什么”风格的编程通常被称为声明式编程。你制定规则，给出了希望实现的目标，让系统来决定如何实现这个目标。它带来的好处非常明显，用这种方式编写的代码更加接近问题陈述了。

13.1.3 为什么要采用函数式编程

函数式编程具体实践了前面介绍的声明式编程（“你只需要使用不相互影响的表达式，描述想要做什么，由系统来选择如何实现”）和无副作用计算。正如我们前面所讨论的，这两个思想能帮助你更容易地构建和维护系统。

同时也请注意，我们在第3章中使用Lambda表达式介绍的内容，即一些语言的特性，比如构造操作和传递行为对于以自然的方式实现声明式编程是必要的，它们能让我们的程序更便于阅读，易于编写。你可以使用`Stream`将几个操作串接在一起，表达一个复杂的查询。这些都是函数式编程语言的特性；我们在14.5节中介绍结合器时会更加深入地介绍这些内容。

为了让你有更直观的感受，我们会结合Java 8介绍这些语言的新特性，现在我们会具体给出函数式编程的定义，以及它在Java语言中的表述。我们希望表达的是，使用函数式编程，你可以实现更加健壮的程序，还不会有任何的副作用。

13.2 什么是函数式编程

对于“什么是函数式编程”这一问题最简化的回答是“它是一种使用函数进行编程的方式”。那什么是函数呢？

我们很容易想象这样一个方法，它接受一个整型和一个浮点型参数，返回一个浮点型的结果——它也有副作用，随着调用次数的增加，它会不断地更新共享变量，如图13-2所示。

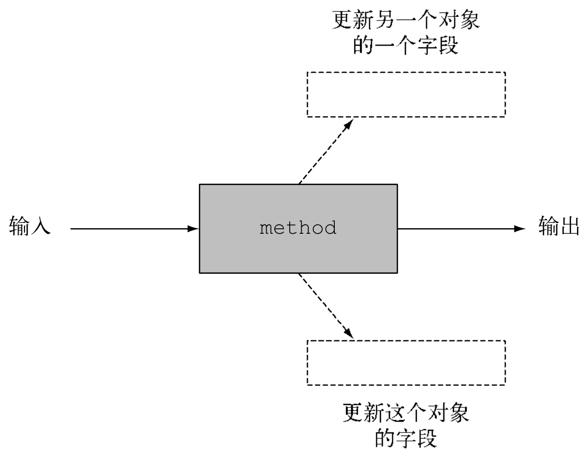


图13-2 带有副作用的函数

在函数式编程的上下文中，一个“函数”对应于一个数学函数：它接受零个或多个参数，生成一个或多个结果，并且不会有任何副作用。你可以把它看成一个黑盒，它接收输入并产生一些输出，如图13-3所示。

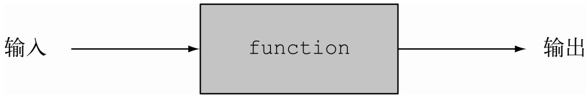


图13-3　一个没有任何副作用的函数

这种类型的函数和你在Java编程语言中见到的函数之间的区别是非常重要的（我们无法想象，log或者sin这样的数学函数会有副作用）。尤其是，使用同样的参数调用数学函数，它所返回的结果一定是相同的。这里，我们暂时不考虑Random.nextInt这样的方法，稍后我们会在介绍引用透明性时讨论这部分内容。

当谈论“函数式”时，我们想说的其实是“像数学函数那样——没有副作用”。由此，编程上的一些精妙问题随之而来。我们的意思是，每个函数都只能使用函数和像if-then-else这样的数学思想来构建吗？或者，我们也允许函数内部执行一些非函数式的操作，只要这些操作的结果不会暴露给系统中的其他部分？换句话说，如果程序有一定的副作用，不过该副作用不会为其他的调用者感知，是否我们能假设这种副作用不存在呢？调用者不需要知道，或者完全不在意这些副作用，因为这对它完全没有影响。

当我们希望能界定这二者之间的区别时，我们将第一种称为纯粹的函数式编程（在本章的最后会讨论这部分内容），后者称为函数式编程。

13.2.1　函数式 Java 编程

编程实战中，你是无法用Java语言以纯粹的函数式来完成一个程序的。比如，Java的I/O模型就包含了带副作用的方法（调用Scanner.nextLine就有副作用，它会从一个文件中读取一行，通常情况两次调用的结果完全不同）。不过，你还是有可能为你系统的核心组件编写接近纯粹函数式的实现。在Java语言中，如果你希望编写函数式的程序，首先需要做的是确保没有人能觉察到你代码的副作用，这也是函数式的含义。假设这样一个函数或者方法，它没有副作用，进入方法体执行时会对一个字段的值加一，退出方法体之前会对该字段减一。对一个单线程的程序而言，这个方法是没有副作用的，可以看作函数式的实现。换个角度而言，如果另一个线程可以查看该字段的值——或者更糟糕的情况，该方法会同时被多个线程并发调用——那么这个方法就不能称之为函数式的实现了。当然，你可以用加锁的方式对方法的方法体进行封装，掩盖这一问题，你甚至可以再次声称该方法符合函数式的约定。但是，这样做之后，你就失去了在你的多核处理器的两个核上并发执行两个方法调用的能力。它的副作用对程序可能是不可见的，不过对于程序员你而言是可见的，因为程序运行的速度变慢了！

我们的准则是，被称为“函数式”的函数或方法都只能修改本地变量。除此之外，它引用的对象都应该是不可修改的对象。通过这种规定，我们期望所有的字段都为final类型，所有的引

用类型字段都指向不可变对象。后续的内容中，你会看到我们实际也允许对方法中全新创建的对象中的字段进行更新，不过这些字段对于其他对象都是不可见的，也不会因为保存对后续调用结果造成影响。

我们前述的准则是不完备的，要成为真正的函数式程序还有一个附加条件，不过它在最初时不太为大家所重视。要被称为函数式，*函数或者方法不应该抛出任何异常*。关于这一点，有一个极为简单而又极为教条的解释：你不应该抛出异常，因为一旦抛出异常，就意味着结果被终止了；不再像我们之前讨论的黑盒模式那样，由`return`返回一个恰当的结果值。不过，这一规则似乎又和我们实际的数学使用有冲突：虽然合法的*数学函数*为每个合法的参数值返回一个确定的结果，很多通用的数学操作在严格意义上称之为*局部函数式*（partial function）可能更为妥当。这种函数对于某些输入值，甚至是大多数的输入值都返回一个确定的结果；不过对另一些输入值，它的结果是*未定义的*，甚至不返回任何结果。这其中一个典型的例子是除法和开平方运算，如果除法的第二操作数是`0`，或者开平方的参数为负数就会发生这样的情况。以Java那样抛出一个异常的方式对这些情况进行建模看起来非常自然。这里存在着一定的争执，有的作者认为抛出代表严重错误的异常是可以接受的，但是捕获异常是一种非函数式的控制流，因为这种操作违背了我们在黑盒模型中定义的“传递参数，返回结果”的规则，引出了代表异常处理的第三支箭头，如图13-4所示。

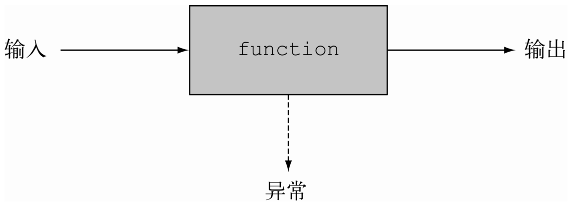


图13-4 抛出一个异常的方法

那么，如果不使用异常，你该如何对除法这样的函数进行建模呢？答案是请使用`Optional<T>`类型：你应该避免让sqrt使用`double sqrt(double)`这样的函数签名，因为这种方式可能抛出异常；与之相反我们推荐你使用`Optional<Double> sqrt(double)`——这种方式下，函数要么返回一个值表示调用成功，要么返回一个对象，表明其无法进行指定的操作。当然，这意味着调用者需要检查方法返回的是否为一个空的`Optional`对象。这件事听起来代价不小，依据我们之前对函数式编程和纯粹的函数式编程的比较，从实际操作的角度出发，你可以选择在本地局部地使用异常，避免通过接口将结果暴露给其他方法，这种方式既取得了函数式的优点，又不会过度膨胀代码。

最后，作为函数式的程序，你的函数或方法调用的库函数如果有副作用，你必须设法隐藏它们的非函数式行为，否则就不能调用这些方法（换句话说，你需要确保它们对数据结构的任何修改对于调用者都是不可见的，你可以通过首次复制，或者捕获任何可能抛出的异常实现这一目的）。在13.2.4节中，你会看到这样的例子，我们通过复制列表的方式，有效地隐藏了方法`insertAll`调用库函数`List.add`所产生的副作用。

13

这些方法通常会使用注释或者使用标记注释声明的方式进行标注——符合我们规定的函数，我们可以将其作为参数传递给并发流处理操作，比如我们在第4~7章介绍过的`Stream.map`方法。

为了各种各样的实战需求，你最终可能会发现即便对函数式的代码，我们还是需要向某些日志文件打印输出调试信息。是的，这意味着严格意义上说，这些代码并非函数式的，但是你已经在实际中享受了函数式程序带来的大多数好处。

13.2.2　引用透明性

“没有可感知的副作用”（不改变对调用者可见的变量、不进行I/O、不抛出异常）的这些限制都隐含着*引用透明性*。如果一个函数只要传递同样的参数值，总是返回同样的结果，那这个函数就是引用透明的。`String.replace`方法就是引用透明的，因为像`"raoul".replace('r', 'R')`这样的调用总是返回同样的结果（`replace`方法返回一个新的字符串，用小写的`r`替换掉所有大写的`R`），而不是更新它的`this`对象，所以它可以被看成函数式的。

换句话说，函数无论在何处、何时调用，如果使用同样的输入总能持续地得到相同的结果，就具备了函数式的特征。这也解释了我们为什么不把`Random.nextInt`看成函数式的方法。Java语言中，使用`Scanner`对象从用户的键盘读取输入也违反了引用透明性原则，因为每次调用`nextLine`时都可能得到不同的结果。不过，将两个`final int`类型的变量相加总能得到同样的结果，因为在这种声明方式下，变量的内容是不会被改变的。

引用透明性是理解程序的一个重要属性。它还包含了对代价昂贵或者需长时间计算才能得到结果的变量值的优化（通过保存机制而不是重复计算），我们通常将其称为*记忆化*或者*缓存*。虽然重要，但是现在讨论还是有些跑题，我们会在14.5节进行介绍。

Java语言中，关于引用透明性还有一个比较复杂的问题。假设你对一个返回列表的方法调用了两次。这两次调用会返回内存中的两个不同列表，不过它们包含了相同的元素。如果这些列表被当作可变的对象值（因此是不相同的），那么该方法就不是引用透明的。如果你计划将这些列表作为单纯的值（不可修改），那么把这些值看成相同的是合理的，这种情况下该方法是引用透明的。通常情况下，*在函数式编程中，你应该选择使用引用透明的函数*。我们会在14.5节继续讨论这一主题。现在我们想探讨从更大的范围看是否应该修改对象的值。

13.2.3　面向对象的编程和函数式编程的对比

我们由函数式编程和（极端）典型的面向对象编程的对比入手进行介绍，最终你会发现Java 8认为这些风格其实只是面向对象的一个极端。作为Java程序员，毫无疑问，你一定使用过某种函数式编程，也一定使用过某些我们称为极端面向对象的编程。正如我们在第1章中所介绍的那样，由于硬件（比如多核）和程序员期望（比如使用类数据库查询式的语言去操纵数据）的变化，促使Java的软件工程风格在某种程度上愈来愈向函数式的方向倾斜，本书的目的之一就是要帮助你应对这种潮流的变化。

关于这个问题有两种观点。一种支持极端的面向对象：任何事物都是对象，程序要么通过更

新字段完成操作，要么调用对与它相关的对象进行更新的方法。另一种观点支持引用透明的函数式编程，认为方法不应该有（对外部可见的）对象修改。实际操作中，Java程序员经常混用这些风格。你可能会使用包含了可变内部状态的迭代器遍历某个数据结构，同时又通过函数式的方式（我们曾经讨论过，可以使用可变局部变量实现这一目标）计算数据结构中的变量之和。本章接下来的一节以及下一章中主要的内容都围绕这函数式编程的技巧展开，帮助你编写更加模块化，更适应多核处理器的应用程序。这些技巧和思想会成为你编程武器库中的秘密武器。

13.2.4 函数式编程实战

让我们从解决一个示例函数式的编程练习题入手：给定一个列表`List<value>`，比如{1, 4, 9}，构造一个`List<List<Integer>>`，它的成员都是类表{1, 4, 9}的子集——我们暂时不考虑元素的顺序。{1, 4, 9}的子集是{1, 4, 9}、{1, 4}、{1, 9}、{4, 9}、{1}、{4}、{9}以及{}。

包括空子集在内，这样的子集总共有8个。每个子集都使用`List<Integer>`表示，这就是答案中期望的`List<List<Integer>>`类型。

通常新手碰到这个问题都会觉得无从下手，对于"{1, 4, 9}的子集可以划分为包含1和不包含1的两部分"也需要特别解释①。不包含`1`的子集很简单就是{4, 9}，包含1的子集可以通过将1插入到{4, 9}的各子集得到。这样我们就能利用Java，以一种简单、自然、自顶向下的函数式编程方式实现该程序了（一个常见的编程错误是认为空的列表没有子集）。

```
static List<List<Integer>> subsets(List<Integer> list) {
    if (list.isEmpty()) {              // 如果输入为空，它就只包含一个子集，既空列表自身
        List<List<Integer>> ans = new ArrayList<>();
        ans.add(Collections.emptyList());
        return ans;
    }
    Integer first = list.get(0);
    List<Integer> rest = list.subList(1,list.size());

    List<List<Integer>> subans = subsets(rest);   // 否则就取出一个元素first，找出剩余部分的所有子集，并将其赋予subans。subans构成了结果的另外一半
    List<List<Integer>> subans2 = insertAll(first, subans);   // 答案的另一半是subans2，它包含了subans中的所有列表，但是经过调整，在每个列表的第一个元素之前添加了first
    return concat(subans, subans2);   // 将两个子答案整合在一起就完成了任务，简单吗？
}
```

如果给出的输入是{1, 4, 9}，程序最终给出的答案是{{}, {9}, {4}, {4, 9}, {1}, {1, 9}, {1, 4}, {1, 4, 9}}。当你完成了缺失的两个方法之后可以实际运行下这个程序。

我们一起回顾下你已经完成了哪些工作。你假设缺失的方法`insertAll`和`concat`自身都是函数式的，并依此推断你的`subsets`方法也是函数式的，因为该方法中没有任何操作会修改现有的结构（如果你熟悉数学的话，你大概对此很熟悉，这就是著名的*归纳法*啊）。

① 偶尔会有些麻烦（机智！）的学生指出另一种解法，这是一种纯粹的代码把戏，它利用二进制来表示数字（Java解决方案的代码分别对应于000,001,010,011,100,101,110,111）。我们告诉这些学生要通过计算得出结果，而不是通过列出所有列表的排列组合；比如以{1,4,9}而言，它就有六种排列组合。

现在，让我们看看如何定义insertAll方法。这是第一个可能出现的坑。假设你已经定义好了insertAll，它会修改传递给它的参数。那么，该程序会以修改subans2同样的方式，错误地修改subans，最终导致答案中莫名地包含了{1, 4, 9}的8个副本。与之相反，你可以像下面这样实现insertAll的功能：

```
static List<List<Integer>> insertAll(Integer first,
                                     List<List<Integer>> lists) {
    List<List<Integer>> result = new ArrayList<>();
    for (List<Integer> list : lists) {
        List<Integer> copyList = new ArrayList<>();
        copyList.add(first);
        copyList.addAll(list);
        result.add(copyList);
    }
    return result;
}
```

复制列表，从而使你有机会对其进行添加操作。即使底层是可变的，你也不应该复制底层的结构（不过Integer底层是不可变的）

注意到了吗？你现在已经创建了一个新的List，它包含了subans的所有元素。你聪明地利用了Integer对象无法修改这一优势，否则你需要为每个元素创建一个副本。由于聚焦于让insertAll像函数式那样地工作，你很自然地将所有的复制操作放到了insertAll中，而不是它的调用者中。

最终，你还需要定义concat方法。这个例子中，我们提供了一个简单的实现，但是我们希望你不要这样使用（我们展示这段代码的目的只是为了便于你比较不同的编程风格）。

```
static List<List<Integer>> concat(List<List<Integer>> a,
                                  List<List<Integer>> b) {
    a.addAll(b);
    return a;
}
```

不过，我们真正建议你采用的是下面这种方式：

```
static List<List<Integer>> concat(List<List<Integer>> a,
                                  List<List<Integer>> b) {
    List<List<Integer>> r = new ArrayList<>(a);
    r.addAll(b);
    return r;
}
```

为什么呢？第二个版本的concat是纯粹的函数式。虽然它在内部会对对象进行修改（向列表r添加元素），但是它返回的结果基于参数却没有修改任何一个传入的参数。与此相反，第一个版本基于这样的事实，执行完concat(subans, subans2)方法调用后，没人需要再次使用subans的值。对于我们定义的subsets，这的确是事实，所以使用简化版本的concat是个不错的选择。不过，这也取决于你如何审视你的时间，你是愿意为定位诡异的缺陷费劲心机耗费时间呢？还是花费些许的代价创建一个对象的副本呢？

无论你怎样解释这个不太纯粹的concat方法，“只会用于第一参数可以被强制覆盖的场景，或者只会使用在这个subsets方法中，任何对subsets的修改都会遵照这一标准进行代码评审”，

一旦将来的某一天，某个人发现这段代码的某些部分可以复用，并且似乎可以工作时，你未来调试的梦魇就开始了。我们会在14.2节继续讨论这一问题。

请牢记：考虑编程问题时，采用函数式的方法，关注函数的输入参数以及输出结果（即你希望做什么），通常比设计阶段的早期就考虑如何做、修改哪些东西要卓有成效得多。我们现在转入介绍更深入的递归，它是函数式编程特别推崇的一种技术，能帮你更深刻地理解“做什么”这一风格。

13.3 递归和迭代

纯粹的函数式编程语言通常不包含像while或者for这样的迭代构造器。为什么呢？因为这种类型的构造器经常隐藏着陷阱，诱使你修改对象。比如，while循环中，循环的条件需要更新；否则循环就一次都不会执行，要么就进入无限循环的状态。但是，很多情况下循环还是非常有用的。我们在前面的介绍中已经声明过，如果没有人能感知的话，函数式也允许进行变更，这意味着我们可以修改局部变量。我们在Java中使用的for-each循环，for(Apple a : apples { }如果用迭代器方式重写，代码如下：

```
Iterator<Apple> it = apples.iterator();
while (it.hasNext()) {
   Apple apple = it.next();
   // ...
}
```

这并不是问题，因为改变发生时，这些变化（包括使用next方法对迭代器状态的改变以及在while循环内部对apple变量的赋值）对于方法的调用方是不可见的。但是，如果使用for-each循环，比如像下面这个搜索算法就会带来问题，因为循环体会对调用方共享的数据结构进行修改：

```
public void searchForGold(List<String> l, Stats stats){
    for(String s: l){
        if("gold".equals(s)){
            stats.incrementFor("gold");
        }
    }
}
```

实际上，对函数式而言，循环体带有一个无法避免的副作用：它会修改stats对象的状态，而这和程序的其他部分是共享的。

由于这个原因，纯函数式编程语言，比如Haskell直接去除了这样的带有副作用的操作！之后你该如何编写程序呢？比较理论的答案是每个程序都能使用无需修改的递归重写，通过这种方式避免使用迭代。使用递归，你可以消除每步都需更新的迭代变量。一个经典的教学问题是用迭代的方式或者递归的方式（假设输入值大于1）编写一个计算阶乘的函数（参数为正数），代码列表如下。

代码清单13-1　迭代式的阶乘计算

```
static int factorialIterative(int n) {
    int r = 1;
    for (int i = 1; i <= n; i++) {
        r *= i;
    }
    return r;
}
```

代码清单13-2　递归式的阶乘计算

```
static long factorialRecursive(long n) {
    return n == 1 ? 1 : n * factorialRecursive(n-1);
}
```

第一段代码展示了标准的基于循环的结构：变量`r`和`i`在每轮循环中都会被更新。第二段代码以更加类数学的形式给出一个递归方法（方法调用自身）的实现。Java语言中，使用递归的形式通常效率都更差一些，我们很快会讨论这方面的内容。

但是，如果你已经仔细阅读过本书的前面章节，一定知道Java 8的`Stream`提供了一种更加简单的方式，用描述式的方法来定义阶乘，代码如下。

代码清单13-3　基于`Stream`的阶乘

```
static long factorialStreams(long n){
    return LongStream.rangeClosed(1, n)
                     .reduce(1, (long a, long b) -> a * b);
}
```

现在，我们回来谈谈效率问题。作为Java的用户，相信你已经意识到函数式程序的狂热支持者们总是会告诉你说，应该使用递归，摒弃迭代。然而，通常而言，执行一次递归式方法调用的开销要比迭代执行单一机器级的分支指令大不少。为什么呢？每次执行`factorialRecursive`方法调用都会在调用栈上创建一个新的栈帧，用于保存每个方法调用的状态（即它需要进行的乘法运算），这个操作会一直指导程序运行直到结束。这意味着你的递归迭代方法会依据它接收的输入成比例地消耗内存。这也是为什么如果你使用一个大型输入执行`factorialRecursive`方法，很容易遭遇`StackOverflowError`异常：

```
Exception in thread "main" java.lang.StackOverflowError
```

这是否意味着递归百无一用呢？当然不是！函数式语言提供了一种方法解决这一问题：尾–调优化（tail-call optimization）。基本的思想是你可以编写阶乘的一个迭代定义，不过迭代调用发生在函数的最后（所以我们说调用发生在尾部）。这种新型的迭代调用经过优化后执行的速度快很多。作为示例，下面是一个阶乘的“尾–递”（tail-recursive）定义。

代码清单13-4　基于“尾–递”的阶乘

```
static long factorialTailRecursive(long n) {
    return factorialHelper(1, n);
}
```

```
static long factorialHelper(long acc, long n) {
    return n == 1 ? acc : factorialHelper(acc * n, n-1);
}
```

方法factorialHelper属于“尾–递”类型的函数，原因是递归调用发生在方法的最后。对比我们前文中factorialRecursive方法的定义，这个方法的最后一个操作是乘以n，从而得到递归调用的结果。

这种形式的递归是非常有意义的，现在我们不需要在不同的栈帧上保存每次递归计算的中间值，编译器能够自行决定复用某个栈帧进行计算。实际上，在factorialHelper的定义中，立即数（阶乘计算的中间结果）直接作为参数传递给了该方法。再也不用为每个递归调用分配单独的栈帧用于跟踪每次递归调用的中间值——通过方法的参数能够直接访问这些值。

图13-5和图13-6解释了使用递归和“尾–递”实现阶乘定义的不同。

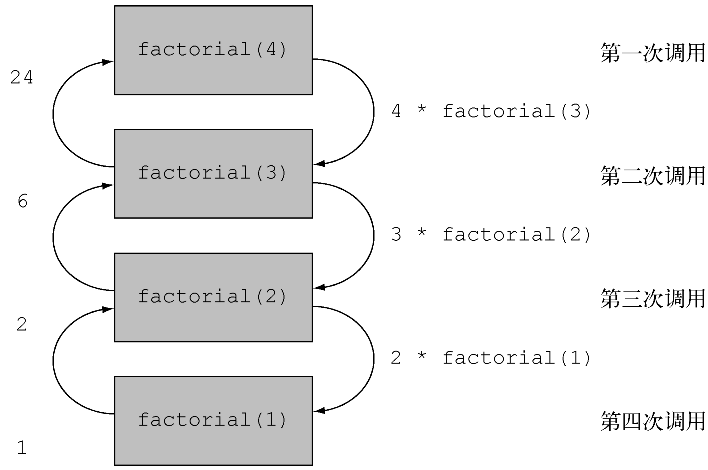


图13-5　使用栈帧方式的阶乘的递归定义

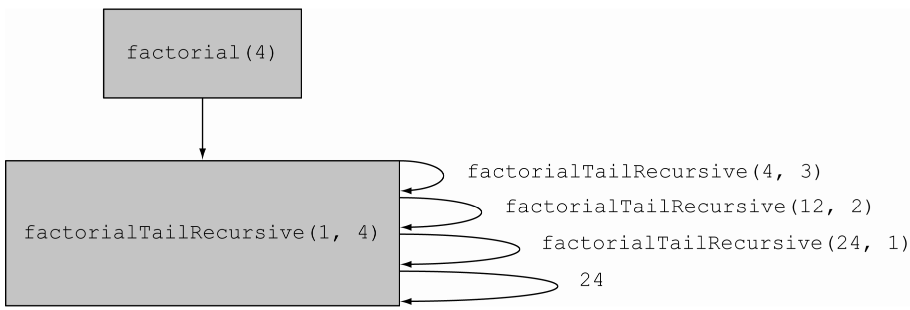


图13-6　阶乘的尾–递定义，这里它只使用了一个栈帧

坏消息是，目前Java还不支持这种优化。但是使用相对于传统的递归，“尾–递”可能是更好的一种方式，因为它为最终实现编译器优化开启了一扇门。很多的现代JVM语言，比如Scala和

Groovy都已经支持对这种形式的递归的优化，最终实现的效果和迭代不相上下（它们的运行速度几乎是相同的）。这意味着坚持纯粹函数式既能享受它的纯净，又不会损失执行的效率。

使用Java 8进行编程时，我们有一个建议，你应该尽量使用Stream取代迭代操作，从而避免变化带来的影响。此外，如果递归能让你以更精炼，并且不带任何副作用的方式实现算法，你就应该用递归替换迭代。实际上，我们看到使用递归实现的例子更加易于阅读，同时又易于实现和理解（比如，我们在前文中展示的子集的例子），大多数时候编程的效率要比细微的执行时间差异重要得多。

这一节，我们讨论了函数式编程，但仅仅是初步介绍了函数式方法的思想——我们介绍的内容甚至适用于最早版本的Java。接下来的一章，我们会讨论Java 8携眷着的一类函数具备了哪些让人耳目一新的强大能力。

13.4　小结

下面是这一章中你应该掌握的关键概念。

- 从长远看，减少共享的可变数据结构能帮助你降低维护和调试程序的代价。
- 函数式编程支持无副作用的方法和声明式编程。
- 函数式方法可以由它的输入参数及输出结果进行判断。
- 如果一个函数使用相同的参数值调用，总是返回相同的结果，那么它是引用透明的。采用递归可以取得迭代式的结构，比如while循环。
- 相对于Java语言中传统的递归，“尾–递”可能是一种更好的方式，它开启了一扇门，让我们有机会最终使用编译器进行优化。

第 14 章 函数式编程的技巧

本章内容

- 一等成员、高阶方法、科里化以及局部应用
- 持久化数据结构
- 生成Java `Stream`时的延迟计算和延迟列表
- 模式匹配以及如何在Java中应用
- 引用透明性和缓存

第13章中，你了解了如何进行函数式的思考；以构造无副作用方法的思想指导你的程序设计能帮助你编写更具维护性的代码。这一章，我们会介绍更高级的函数式编程技巧。你可以将本章看作实战技巧和学术知识的大杂烩，它既包含了能直接用于代码编写的技巧，也包含了能让你知识更渊博的学术信息。我们会讨论高阶函数、科里化、持久化数据结构、延迟列表、模式匹配、具备引用透明性的缓存，以及结合器。

14.1 无处不在的函数

第13章中我们使用术语“函数式编程”意指函数或者方法的行为应该像“数学函数”一样——没有任何副作用。对于使用函数式语言的程序员而言，这个术语的范畴更加宽泛，它还意味着函数可以像任何其他值一样随意使用：可以作为参数传递，可以作为返回值，还能存储在数据结构中。能够像普通变量一样使用的函数称为一等*函数*（first-class function）。这是Java 8补充的全新内容：通过`::`操作符，你可以创建一个*方法引用*，像使用函数值一样使用方法，也能使用Lambda表达式（比如，`(int x) -> x + 1`）直接表示方法的值。Java 8中使用下面这样的方法引用将一个方法引用保存到一个变量是合理合法的：

```
Function<String, Integer> strToInt = Integer::parseInt;
```

14.1.1 高阶函数

目前为止，我们使用函数值属于一等这个事实只是为了将它们传递给Java 8的流处理操作（正如我们在第4~7章看到的一样），达到行为参数化的效果，类似我们在第1章和第2章中将

Apple::isGreenApple作为参数值传递给filterApples方法那样。但这仅仅是个开始。另一个有趣的例子是静态方法Comparator.comparing的使用，它接受一个函数作为参数同时返回另一个函数（一个比较器），代码如下所示。图14-1对这段逻辑进行了解释。

```
Comparator<Apple> c = comparing(Apple::getWeight);
```

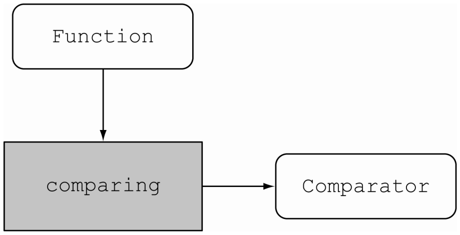


图14-1　comparing方法接受一个函数作为参数，同时返回另一个函数

第3章我们构造函数创建流水线时，做了一些类似的事：

```
Function<String, String> transformationPipeline
    = addHeader.andThen(Letter::checkSpelling)
              .andThen(Letter::addFooter);
```

函数式编程的世界里，如果函数，比如Comparator.comparing，能满足下面任一要求就可以被称为高阶函数（higher-order function）：

- 接受至少一个函数作为参数
- 返回的结果是一个函数

这些都和Java 8直接相关。因为Java 8中，函数不仅可以作为参数传递，还可以作为结果返回，能赋值给本地变量，也可以插入到某个数据结构。比如，一个计算口袋的程序可能有这样的一个Map<String, Function<Double, Double>>，它将字符串sin映射到方法Function<Double, Double>，实现对Math::sin的方法引用。我们在第8章介绍工厂方法时进行了类似的操作。

对于喜欢第3章结尾的那个微积分示例的读者，由于它接受一个函数作为参数（比如，(Double x) -> x * x），又返回一个函数作为结果（这个例子中返回值是(Double x) -> 2 * x），你可以用不同的方式实现类型定义，如下所示：

```
Function<Function<Double,Double>, Function<Double,Double>>
```

我们把它定义成Function类型（最左边的Function），目的是想显式地向你确认可以将这个函数传递给另一个函数。但是，最好使用差异化的类型定义，函数签名如下：

```
Function<Double,Double> differentiate(Function<Double,Double> func)
```

其实二者说的是同一件事。

副作用和高阶函数

第7章中我们了解到传递给流操作的函数应该是无副作用的，否则会发生各种各样的问题

（比如错误的结果，有时由于竞争条件甚至会产生我们无法预期的结果）。这一原则在你使用高阶函数时也同样适用。编写高阶函数或者方法时，你无法预知会接收什么样的参数——一旦传入的参数有某些副作用，我们将会一筹莫展！如果作为参数传入的函数可能对你程序的状态产生某些无法预期的改变，一旦发生问题，你将很难理解程序中发生了什么；它们甚至会用某种难于调试的方式调用你的代码。因此，将所有你愿意接收的作为参数的函数可能带来的副作用以文档的方式记录下来是一个不错的设计原则，最理想的情况下你接收的函数参数应该没有任何副作用！

现在我们转向讨论科里化：它是一种可以帮助你模块化函数、提高代码重用性的技术。

14.1.2　科里化

给出科里化的理论定义之前，让我们先来看一个例子。应用程序通常都会有国际化的需求，将一套单位转换到另一套单位是经常碰到的问题。

单位转换通常都会涉及转换因子以及基线调整因子的问题。比如，将摄氏度转换到华氏度的公式是`CtoF(x) = x*9/5 + 32`。

所有的单位转换几乎都遵守下面这种模式：

(1) 乘以转换因子

(2) 如果需要，进行基线调整

你可以使用下面这段通用代码表达这一模式：

```
static double converter(double x, double f, double b) {
    return x * f + b;
}
```

这里`x`是你希望转换的数量，`f`是转换因子，`b`是基线值。但是这个方法有些过于宽泛了。通常，你还需要在同一类单位之间进行转换，比如公里和英里。当然，你也可以在每次调用`converter`方法时都使用3个参数，但是每次都提供转换因子和基准比较繁琐，并且你还极有可能输入错误。

当然，你也可以为每一个应用编写一个新方法，不过这样就无法对底层的逻辑进行复用。

这里我们提供一种简单的解法，它既能充分利用已有的逻辑，又能让`converter`针对每个应用进行定制。你可以定义一个“工厂”方法，它生产带一个参数的转换方法，我们希望借此来说明科里化。下面是这段代码：

```
static DoubleUnaryOperator curriedConverter(double f, double b){
    return (double x) -> x * f + b;
}
```

14

现在，你要做的只是向它传递转换因子和基准值（`f`和`b`），它会不辞辛劳地按照你的要求返回一个方法（使用参数`x`）。比如，你现在可以按照你的需求使用工厂方法产生你需要的任何`converter`：

```
DoubleUnaryOperator convertCtoF = curriedConverter(9.0/5, 32);
DoubleUnaryOperator convertUSDtoGBP = curriedConverter(0.6, 0);
DoubleUnaryOperator convertKmtoMi = curriedConverter(0.6214, 0);
```

由于DoubleUnaryOperator定义了方法applyAsDouble，你可以像下面这样使用你的converter：

```
double gbp = convertUSDtoGBP.applyAsDouble(1000);
```

这样一来，你的代码就更加灵活了，同时它又复用了现有的转换逻辑！让我们一起回顾下你都做了哪些工作。你并没有一次性地向converter方法传递所有的参数x、f和b，相反，你只是使用了参数f和b并返回了另一个方法，这个方法会接收参数x，最终返回你期望的值x * f + b。通过这种方式，你复用了现有的转换逻辑，同时又为不同的转换因子创建了不同的转换方法。

科里化的理论定义

科里化[①]是一种将具备2个参数（比如，x和y）的函数f转化为使用一个参数的函数g，并且这个函数的返回值也是一个函数，它会作为新函数的一个参数。后者的返回值和初始函数的返回值相同，即f(x,y) = (g(x))(y)。

当然，我们可以由此推出：你可以将一个使用了6个参数的函数科里化成一个接受第2、4、6号参数，并返回一个接受5号参数的函数，这个函数又返回一个接受剩下的第1号和第3号参数的函数。

一个函数使用所有参数仅有部分被传递时，通常我们说这个函数是**部分应用的**（partially applied）。

现在我们转而讨论函数式编程的另一个方面。如果你不能修改数据结构，还能用它们编程吗？

14.2　持久化数据结构

这一节中，我们会探讨函数式编程中如何使用数据结构。这一主题有各种名称，比如函数式数据结构、不可变数据结构，不过最常见的可能还要算持久化数据结构（不幸的是，这一术语和数据库中的持久化概念有一定的冲突，数据库中它代表的是“生命周期比程序的执行周期更长的数据”）。

我们应该注意的第一件事是，函数式方法不允许修改任何全局数据结构或者任何作为参数传入的参数。为什么呢？因为一旦对这些数据进行修改，两次相同的调用就很可能产生不同的结构——这违背了引用透明性原则，我们也就无法将方法简单地看作由参数到结果的映射。

① 科里化的概念最早由俄国数学家Moses Schönfinkel引入，而后由著名的数理逻辑学家哈斯格尔·科里（Haskell Curry）丰富和发展，科里化由此得名。它表示一种将一个带有n元组参数的函数转换成n个一元函数链的方法。——译者注

14.2.1 破坏式更新和函数式更新的比较

让我们看看不这么做会导致怎样的结果。假设你需要使用一个可变类TrainJourney（利用一个简单的单向链接列表实现）表示从A地到B地的火车旅行，你使用了一个整型字段对旅程的一些细节进行建模，比如当前路途段的价格。旅途中你需要换乘火车，所以需要使用几个由onward字段串联在一起的TrainJourney对象；直达火车或者旅途最后一段对象的onward字段为null：

```
class TrainJourney {
    public int price;
    public TrainJourney onward;
    public TrainJourney(int p, TrainJourney t) {
        price = p;
        onward = t;
    }
}
```

假设你有几个相互分隔的TrainJourney对象分别代表从X到Y和从Y到Z的旅行。你希望创建一段新的旅行，它能将两个TrainJourney对象串接起来（即从X到Y再到Z）。

一种方式是采用简单的传统命令式的方法将这些火车旅行对象链接起来，代码如下：

```
static TrainJourney link(TrainJourney a, TrainJourney b){
    if (a==null) return b;
    TrainJourney t = a;
    while(t.onward != null){
        t = t.onward;
    }
    t.onward = b;
    return a;
}
```

这个方法是这样工作的，它找到TrainJourney对象a的下一站，将其由表示a列表结束的null替换为列表b（如果a不包含任何元素，你需要进行特殊处理）。

这就出现了一个问题：假设变量firstJourney包含了从X地到Y地的线路，另一个变量secondJourney包含了从Y地到Z地的线路。如果你调用link(firstJourney, secondJourney)方法，这段代码会破坏性地更新firstJourney，结果secondJourney也会加被入到firstJourney，最终请求从X地到Z地的用户会如其所愿地看到整合之后的旅程，不过从X地到Y地的旅程也被破坏性地更新了。这之后，变量firstJourney就不再代表从X到Y的旅程，而是一个新的从X到Z的旅程了！这一改动会导致依赖原先的firstJourney代码失效！假设firstJourney表示的是清晨从伦敦到布鲁塞尔的火车，这趟车上后一段的乘客本来打算要去布鲁塞尔，可是发生这样的改动之后他们莫名地多走了一站，最终可能跑到了科隆。现在你大致了解了数据结构修改的可见性会导致怎样的问题了，作为程序员，我们一直在与这种缺陷作斗争。

函数式编程解决这一问题的方法是禁止使用带有副作用的方法。如果你需要使用表示计算结果的数据结果，那么请创建它的一个副本而不要直接修改现存的数据结构。这一最佳实践也适用于标准的面向对象程序设计。不过，对这一原则，也存在着一些异议，比较常见的是认为这样做

会导致过度的对象复制，有些程序员会说“我会记住那些有副作用的方法”或者“我会将这些写入文档”。但这些都不能解决问题，这些坑都留给了接受代码维护工作的程序员。采用函数式编程方案的代码如下：

```
static TrainJourney append(TrainJourney a, TrainJourney b){
    return a==null ? b : new TrainJourney(a.price, append(a.onward, b));
}
```

很明显，这段代码是函数式的（它没有做任何修改，即使是本地的修改），它没有改动任何现存的数据结构。不过，也请特别注意，这段代码有一个特别的地方，它并未创建整个新TrainJourney对象的副本——如果a是n个元素的序列，b是m个元素的序列，那么调用这个函数后，它返回的是一个由$n+m$个元素组成的序列，这个序列的前n个元素是新创建的，而后m个元素和TrainJourney对象b是共享的。另外，也请注意，用户需要确保不对append操作的结果进行修改，因为一旦这样做了，作为参数传入的TrainJourney对象序列b就可能被破坏。图14-2和图14-3解释说明了破坏式append和函数式append之间的区别。

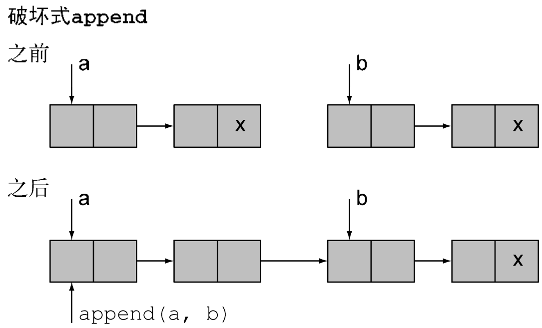


图14-2　以破坏式更新的数据结构

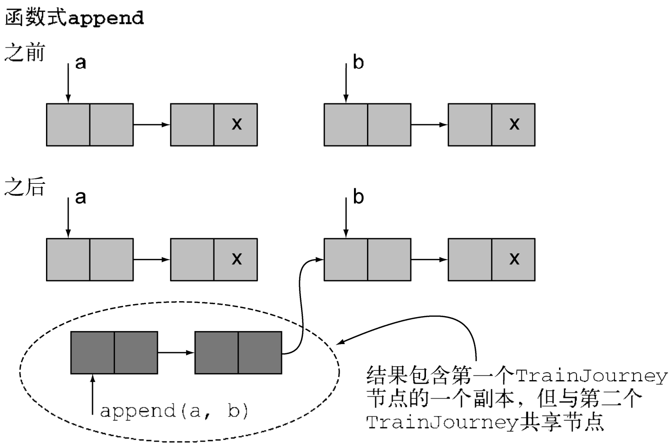


图14-3　函数式，不会对原有数据结构进行改动

14.2.2 另一个使用 Tree 的例子

转入新主题之前，让我们再看一个使用其他数据结构的例子——我们想讨论的对象是二叉查找树，它也是HashMap实现类似接口的方式。我们的设计中Tree包含了String类型的键，以及int类型的键值，它可能是名字或者年龄：

```
class Tree {
   private String key;
   private int val;
   private Tree left, right;
   public Tree(String k, int v, Tree l, Tree r) {
     key = k; val = v; left = l; right = r;
   }
}
class TreeProcessor {
    public static int lookup(String k, int defaultval, Tree t) {
        if (t == null) return defaultval;
        if (k.equals(t.key)) return t.val;
        return lookup(k, defaultval,
                         k.compareTo(t.key) < 0 ? t.left : t.right);
    }
    // 处理Tree的其他方法
}
```

你希望通过二叉查找树找到String值对应的整型数。现在，我们想想你该如何更新与某个键对应的值（简化起见，我们假设键已经存在于这个树中了）：

```
public static void update(String k, int newval, Tree t) {
    if (t == null) { /* 应增加一个新的节点 */ }
    else if (k.equals(t.key)) t.val = newval;
    else update(k, newval, k.compareTo(t.key) < 0 ? t.left : t.right);
}
```

对这个例子，增加一个新的节点会复杂很多；最简单的方法是让update直接返回它刚遍历的树（除非你需要加入一个新的节点，否则返回的树结构是不变的）。现在，这段代码看起来已经有些臃肿了（因为update试图对树进行原地更新，它返回的是跟传入的参数同样的树，但是如果最初的树为空，那么新的节点会作为结果返回）。

```
public static Tree update(String k, int newval, Tree t) {
    if (t == null)
       t = new Tree(k, newval, null, null);
    else if (k.equals(t.key))
       t.val = newval;
    else if (k.compareTo(t.key) < 0)
       t.left = update(k, newval, t.left);
    else
       t.right = update(k, newval, t.right);
    return t;
}
```


注意，这两个版本的update都会对现有的树进行修改，这意味着使用树存放映射关系的所

有用户都会感知到这些修改。

14.2.3　采用函数式的方法

那么这一问题如何通过函数式的方法解决呢？你需要为新的键–值对创建一个新的节点，除此之外你还需要创建从树的根节点到新节点的路径上的所有节点。通常而言，这种操作的代价并不太大，如果树的深度为d，并且保持一定的平衡性，那么这棵树的节点总数是2^d，这样你就只需要重新创建树的一小部分节点了。

```
public static Tree fupdate(String k, int newval, Tree t) {
    return (t == null) ?
        new Tree(k, newval, null, null) :
          k.equals(t.key) ?
            new Tree(k, newval, t.left, t.right) :
          k.compareTo(t.key) < 0 ?
            new Tree(t.key, t.val, fupdate(k,newval, t.left), t.right) :
            new Tree(t.key, t.val, t.left, fupdate(k,newval, t.right));
}
```

这段代码中，我们通过一行语句进行的条件判断，没有采用if-then-else这种方式，目的是希望强调一个思想，那就是该函数体仅包含一条语句，没有任何副作用。不过你也可以按照自己的习惯，使用if-then-else这种方式，在每一个判断结束处使用return返回。

那么，update 和fupdate之间的区别到底是什么呢？我们注意到，前文中方法update有这样一种假设，即每一个update的用户都希望共享同一份数据结构，也希望能了解程序任何部分所做的更新。因此，无论任何时候，只要你使用非函数式代码向树中添加某种形式的数据结构，请立刻创建它的一份副本，因为谁也不知道将来的某一天，某个人会突然对它进行修改，这一点非常重要（不过也经常被忽视）。与之相反，fupdate是纯函数式的。它会创建一个新的树，并将其作为结果返回，通过参数的方式实现共享。图14-4对这一思想进行了阐释。你使用了一个树结构，树的每个节点包含了person对象的姓名和年龄。调用fupdate不会修改现存的树，它会在原有树的一侧创建新的节点，同时保证不损坏现有的数据结构。

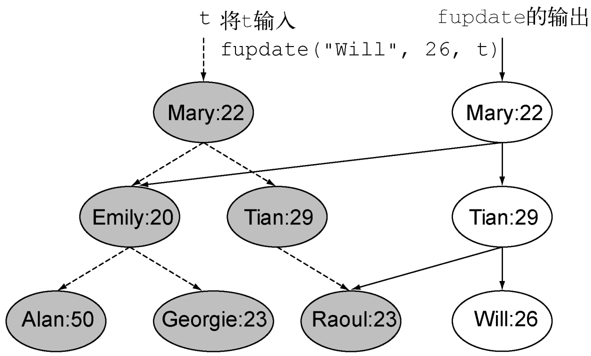


图14-4　对树结构进行更新时，现存数据结构不会被破坏

这种函数式数据结构通常被称为持久化的——数据结构的值始终保持一致，不受其他部分变化的影响——这样，作为程序员的你才能确保`fupdate`不会对作为参数传入的数据结构进行修改。不过要达到这一效果还有一个附加条件：这个约定的另一面是，所有使用持久化数据结构的用户都必须遵守这一“不修改”原则。如果不这样，忽视这一原则的程序员很有可能修改`fupdate`的结果（比如，修改Emily的年纪为20岁）。这会成为一个例外（也是我们不期望发生的）事件，为所有使用该结构的方法感知，并在之后修改作为参数传递给`fupdate`的数据结构。

通过这些介绍，我们了解到`fupdate`可能有更加高效的方式：基于“不对现存结构进行修改”规则，对仅有细微差别的数据结构（比如，用户A看到的树结构与用户B看到的就相差不多），我们可以考虑对这些通用数据结构使用共享存储。你可以凭借编译器，将`Tree`类的字段`key`、`val`、`left`以及`right`声明为`final`执行，“禁止对现存数据结构的修改”这一规则；不过我们也需要注意`final`只能应用于类的字段，无法应用于它指向的对象，如果你想要对对象进行保护，你需要将其中的字段声明为`final`，以此类推。

噢，你可能会说：“我希望对树结构的更新对某些用户可见（当然，这句话的潜台词是其他人看不到这些更新）。”那么，要实现这一目标，你可以通过两种方式：第一种是典型的Java解决方案（对对象进行更新时，你需要特别小心，慎重地考虑是否需要在改动之前保存对象的一份副本）。另一种是函数式的解决方案：逻辑上，你在做任何改动之前都会创建一份新的数据结构（这样一来就不会有任何的对象发生变更），只要确保按照用户的需求传递给他正确版本的数据结构就好了。这一想法甚至还可以通过API直接强制实施。如果数据结构的某些用户需要进行可见性的改动，它们应该调用API，返回最新版的数据结构。对于另一些客户应用，它们不希望发生任何可见的改动（比如，需要长时间运行的统计分析程序），就直接使用它们保存的备份，因为它知道这些数据不会被其他程序修改。

有些人可能会说这个过程很像更新刻录光盘上的文件，刻录光盘时，一个文件只能被激光写入一次，该文件的各个版本分别被存储在光盘的各个位置（智能光盘编辑软件甚至会共享多个不同版本之间的相同部分），你可以通过传递文件起始位置对应的块地址（或者名字中编码了版本信息的文件名）选择你希望使用哪个版本的文件。Java中，情况甚至比刻录光盘还好很多，不再使用的老旧数据结构会被Java虚拟机自动垃圾回收掉。

14.3 Stream 的延迟计算

通过前一章的介绍，你已经了解Stream是处理数据集合的利器。不过，由于各种各样的原因，包括实现时的效率考量，Java 8的设计者们在将Stream引入时采取了比较特殊的方式。其中一个比较显著的局限是，你无法声明一个递归的Stream，因为Stream仅能使用一次。在接下来的一节，我们会详细展开介绍这一局限会带来的问题。

14.3.1 自定义的 Stream

让我们一起回顾下第6章中生成质数的例子，这个例子有助于我们理解递归式Stream的思想。

你大概已经看到，作为MyMathUtils类的一部分，你可以用下面这种方式计算得出由质数构成的Stream：

```
public static Stream<Integer> primes(int n) {
    return Stream.iterate(2, i -> i + 1)
                 .filter(MyMathUtils::isPrime)
                 .limit(n);
}

public static boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

不过这一方案看起来有些笨拙：你每次都需要遍历每个数字，查看它能否被候选数字整除（实际上，你只需要测试那些已经被判定为质数的数字）。

理想情况下，Stream应该实时地筛选掉那些能被质数整除的数字。这听起来有些异想天开，不过我们一起看看怎样才能达到这样的效果。

(1) 你需要一个由数字构成的Stream，你会在其中选择质数。

(2) 你会从该Stream中取出第一个数字（即Stream的首元素），它是一个质数（初始时，这个值是2）。

(3) 紧接着你会从Stream的尾部开始，筛选掉所有能被该数字整除的元素。

(4) 最后剩下的结果就是新的Stream，你会继续用它进行质数的查找。本质上，你还会回到第一步，继续进行后续的操作，所以这个算法是递归的。

注意，这个算法不是很好，原因是多方面的[①]。不过，就说明如何使用Stream展开工作这个目的而言，它还是非常合适的，因为算法简单，容易说明。让我们试着用Stream API对这个算法进行实现。

1. 第一步： 构造由数字组成的Stream

你可以使用方法IntStream.iterate构造由数字组成的Stream，它由2开始，可以上达无限，就像我们在第5章中介绍的那样，代码如下：

```
static Intstream numbers(){
    return IntStream.iterate(2, n -> n + 1);
}
```

2. 第二步： 取得首元素

IntStream类提供了方法findFirst，可以返回Stream的第一个元素：

```
static int head(IntStream numbers){
    return numbers.findFirst().getAsInt();
}
```

① 关于为什么这个算法很糟糕的更多信息，请参考http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf。

3. 第三步：对尾部元素进行筛选

定义一个方法取得Stream的尾部元素：

```
static IntStream tail(IntStream numbers){
    return numbers.skip(1);
}
```

拿到Stream的头元素，你可以像下面这段代码那样对数字进行筛选：

```
IntStream numbers = numbers();
int head = head(numbers);
IntStream filtered = tail(numbers).filter(n -> n % head != 0);
```

4. 第四步：递归地创建由质数组成的Stream

现在到了最复杂的部分。你可能试图将筛选返回的Stream作为参数再次传递给该方法，这样你可以接着取得它的头元素，继续筛选掉更多的数字，如下所示：

```
static IntStream primes(IntStream numbers) {
    int head = head(numbers);
    return IntStream.concat(
             IntStream.of(head),
             primes(tail(numbers).filter(n -> n % head != 0))
           );
}
```

5. 坏消息

不幸的是，如果执行步骤四中的代码，你会遭遇如下这个错误："java.lang.IllegalStateException: stream has already been operated upon or closed." 实际上，你正试图使用两个终端操作：`findFirst`和`skip`将Stream切分成头尾两部分。还记得我们在第4章中介绍的内容吗？一旦你对Stream执行一次终端操作调用，它就永久地终止了！

6. 延迟计算

除此之外，该操作还附带着一个更为严重的问题：静态方法`IntStream.concat`接受两个Stream实例作参数。但是，由于第二个参数是`primes`方法的直接递归调用，最终会导致出现无限递归的状况。然而，对大多数的Java应用而言，Java 8在Stream上的这一限制，即"不允许递归定义"是完全没有影响的，使用Stream后，数据库的查询更加直观了，程序还具备了并发的能力。所以，Java 8的设计者们进行了很好的平衡，选择了这一皆大欢喜的方案。不过，Scala和Haskell这样的函数式语言中Stream所具备的通用特性和模型仍然是你编程武器库中非常有益的补充。你需要一种方法推迟`primes`中对`concat`的第二个参数计算。如果用更加技术性的程序设计术语来描述，我们称之为*延迟计算*、*非限制式计算*或者*名调用*。只在你需要处理质数的那个时刻（比如，要调用方法`limit`了）才对Stream进行计算。Scala（我们会在下一章介绍）提供了对这种算法的支持。在Scala中，你可以用下面的方式重写前面的代码，操作符`#::`实现了延迟连接的功能（只有在你实际需要使用Stream时才对其进行计算）：


```
def numbers(n: Int): Stream[Int] = n #:: numbers(n+1)

def primes(numbers: Stream[Int]): Stream[Int] = {
```

```
    numbers.head #:: primes(numbers.tail filter (n -> n % numbers.head != 0))
}
```

看不懂这段代码？完全没关系。我们展示这段代码的目的只是希望能让你了解Java和其他的函数式编程语言的区别。让我们一起回顾一下刚刚介绍的参数是如何计算的，这对我们后面的内容很有裨益。在Java语言中，你执行一次方法调用时，传递的所有参数在第一时间会被立即计算出来。但是，在Scala中，通过`#::`操作符，连接操作会立刻返回，而元素的计算会推迟到实际计算需要的时候才开始。现在，让我们看看如何通过Java实现延迟列表的思想。

14.3.2　创建你自己的延迟列表

Java 8的Stream以其延迟性而著称。它们被刻意设计成这样，即延迟操作，有其独特的原因：Stream就像是一个黑盒，它接收请求生成结果。当你向一个 Stream发起一系列的操作请求时，这些请求只是被一一保存起来。只有当你向Stream发起一个终端操作时，才会实际地进行计算。这种设计具有显著的优点，特别是你需要对Stream进行多个操作时（你有可能先要进行`filter`操作，紧接着做一个`map`，最后进行一次终端操作`reduce`）；这种方式下Stream只需要遍历一次，不需要为每个操作遍历一次所有的元素。

这一节，我们讨论的主题是延迟列表，它是一种更加通用的Stream形式（延迟列表构造了一个跟Stream非常类似的概念）。延迟列表同时还提供了一种极好的方式去理解高阶函数；你可以将一个函数作为值放置到某个数据结构中，大多数时候它就静静地待在那里，一旦对其进行调用（即根据需要），它能够创建更多的数据结构。图14-5解释了这一思想。

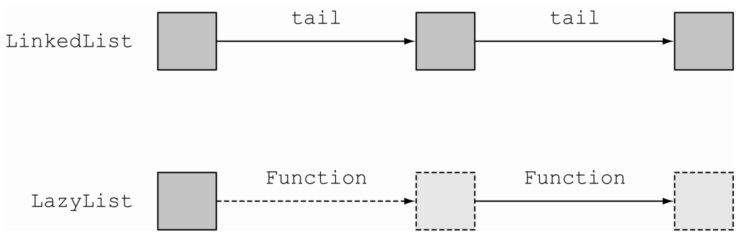


图14-5　`LinkedList`的元素存在于（并不断延展）内存中。而`LazyList`的元素由函数在需要使用时动态创建，你可以将它们看成实时延展的

我们谈论得已经很多，现在让我们一起看看它是如何工作的。你想要利用我们前面介绍的算法，生成一个由质数构成的无限列表。

1. 一个基本的链接列表

还记得吗，你可以通过下面这种方式，用Java语言实现一个简单的名为`MyLinkedList`的链接–列表–式的类（这里我们只考虑最精简的`MyList`接口）：

```
interface MyList<T> {
    T head();

    MyList<T> tail();
```

```
    default boolean isEmpty() {
        return true;
    }
}

class MyLinkedList<T> implements MyList<T> {
    private final T head;
    private final MyList<T> tail;
    public MyLinkedList(T head, MyList<T> tail) {
        this.head = head;
        this.tail = tail;
    }

    public T head() {
        return head;
    }

    public MyList<T> tail() {
        return tail;
    }

    public boolean isEmpty() {
        return false;
    }
}

class Empty<T> implements MyList<T> {

    public T head() {
        throw new UnsupportedOperationException();
    }
    public MyList<T> tail() {
        throw new UnsupportedOperationException();
    }
}
```

你现在可以构造一个示例的`MyLinkedList`值，如下所示：

```
MyList<Integer> l =
    new MyLinkedList<>(5, new MyLinkedList<>(10, new Empty<>()));
```

2. 一个基础的延迟列表

对这个类进行改造，使其符合延迟列表的思想，最简单的方法是避免让`tail`立刻出现在内存中，而是像第3章那样，提供一个`Supplier<T>`方法（你也可以将其看成一个使用函数描述符`void -> T`的工厂方法），它会产生列表的下一个节点。使用这种方式的代码如下：

```
import java.util.function.Supplier;
class LazyList<T> implements MyList<T>{
    final T head;
    final Supplier<MyList<T>> tail;
    public LazyList(T head, Supplier<MyList<T>> tail) {
        this.head = head;
```

```
        this.tail = tail;
    }

    public T head() {
        return head;
    }

    public MyList<T> tail() {
        return tail.get();
    }

    public boolean isEmpty() {
        return false;
    }
}
```

注意，与前面的head不同，这里tail使用了一个Supplier方法提供了延迟性

调用Supplier的get方法会触发延迟列表（LazyList）的节点创建，就像工厂会创建新的对象一样。

现在，你可以像下面那样传递一个Supplier作为LazyList的构造器的tail参数，创建由数字构成的无限延迟列表了，该方法会创建一系列数字中的下一个元素：

```
public static LazyList<Integer> from(int n) {
    return new LazyList<Integer>(n, () -> from(n+1));
}
```

如果尝试执行下面的代码，你会发现，下面的代码执行会打印输出“2 3 4”。这些数字真真实实都是实时计算得出的。你可以在恰当的位置插入System.out.println进行查看，如果from(2)执行得很早，它试图计算从2!开始的所有数字，它会永远运行下去，这时你不需要做任何事情。

```
LazyList<Integer> numbers = from(2);
int two = numbers.head();
int three = numbers.tail().head();
int four = numbers.tail().tail().head();

System.out.println(two + " " + three + " " + four);
```

3. 回到生成质数

看看你能否利用我们目前已经做的去生成一个自定义的质数延迟列表（有些时候，你会遭遇无法使用Stream API的情况）。如果你将之前使用Stream API的代码转换成使用我们新版的LazyList，它看起来会像下面这段代码：

```
public static MyList<Integer> primes(MyList<Integer> numbers) {
    return new LazyList<>(
                numbers.head(),
                () -> primes(
                        numbers.tail()
                                .filter(n -> n % numbers.head() != 0)
                            )
    );
}
```

4. 实现一个延迟筛选器

不过，这个LazyList（更确切地说是List接口）并未定义filter方法，所以前面的这段代码是无法编译通过的。让我们添加该方法的一个定义，修复这个问题：

```
public MyList<T> filter(Predicate<T> p) {
    return isEmpty() ?
            this :
            p.test(head()) ?
                new LazyList<>(head(), () -> tail().filter(p)) :
                tail().filter(p);
}
```

你可以返回一个新的Empty<>()，不过这和返回一个空对象的效果是一样的

你的代码现在可以通过编译，准备使用了。通过链接对tail和head的调用，你可以计算出头三个质数：

```
LazyList<Integer> numbers = from(2);
int two = primes(numbers).head();
int three = primes(numbers).tail().head();
int five = primes(numbers).tail().tail().head();

System.out.println(two + " " + three + " " + five);
```

这段代码的输出是“2 3 5”，这是头三个质数的值。现在，你可以把玩这段程序了，比如，你可以打印输出所有的质数（printAll方法会递归地打印输出列表的头尾元素，这个程序会永久地运行下去）：

```
static <T> void printAll(MyList<T> list){
    while (!list.isEmpty()){
        System.out.println(list.head());
        list = list.tail();
    }
}
printAll(primes(from(2)));
```

本章的主题是函数式编程，我们应该在更早的时候就让你知道其实有更加简洁地方式完成这一递归操作：

```
static <T> void printAll(MyList<T> list){
    if (list.isEmpty())
        return;
    System.out.println(list.head());
    printAll(list.tail());
}
```

但是，这个程序不会永久地运行下去；它最终会由于栈溢出而失效，因为Java不支持尾部调用消除（tail call elimination），这一点我们曾经在第13章介绍过。

5. 何时使用

到目前为止，你已经构建了大量技术，包括延迟列表和函数，使用它们却只定义了一个包含质数的数据结构。为什么呢？哪些实际的场景可以使用这些技术呢？好吧，你已经了解了如何向

数据结构中插入函数（因为Java 8允许你这么做），这些函数可以用于按需创建数据结构的一部分，现在你不需要在创建数据结构时就一次性地定义所有的部分。如果你在编写游戏程序，比如棋牌类游戏，你可以定义一个数据结构，它在形式上涵盖了由所有可能移动构成的一个树（这些步骤要在早期完成计算工作量太大），具体的内容可以在运行时创建。最终的结果是一个延迟树，而不是一个延迟列表。我们本章关注延迟列表，原因是它可以和Java 8的另一个新特性Stream串接起来，我们能够针对性地讨论Stream和延迟列表各自的优缺点。

还有一个问题就是性能。我们很容易得出结论，延迟操作的性能会比提前操作要好——仅在程序需要时才计算值和数据结构当然比传统方式下一次性地创建所有的值（有时甚至比实际需求更多的值）要好。不过，实际情况并非如此简单。完成延迟操作的开销，比如`LazyList`中每个元素之间执行额外`Suppliers`调用的开销，有可能超过你猜测会带来的好处，除非你仅仅只访问整个数据结构的10%，甚至更少。最后，还有一种微妙的方式会导致你的`LazyList`并非真正的延迟计算。如果你遍历`LazyList`中的值，比如`from(2)`，可能直到第10个元素，这种方式下，它会创建每个节点两次，最终创建20个节点，而不是10个。这几乎不能被称为延迟计算。问题在于每次实时访问`LazyList`的元素时，`tail`中的`Supplier`都会被重复调用；你可以设定`tail`中的`Supplier`方法仅在第一次实时访问时才执行调用，从而修复这一问题——计算的结果会缓存起来——效果上对列表进行了增强。要实现这一目标，你可以在`LazyList`的定义中添加一个私有的`Optional<LazyList<T>>`类型字段`alreadyComputed`，`tail`方法会依据情况查询及更新该字段的值。纯函数式语言Haskell就是以这种方式确保它所有的数据结构都恰当地进行了延迟。如果你对这方面的细节感兴趣，可以查看相关文章。[①]

我们推荐的原则是将延迟数据结构作为你编程兵器库中的强力武器。如果它们能让程序设计更简单，就尽量使用它们。如果它们会带来无法接受的性能损失，就尝试以更加传统的方式重新实现它们。

现在，让我们转向几乎所有函数式编程语言中都提供的一个特性，不过Java语言中暂时并未提供这一特性，它就是*模式匹配*。

14.4　模式匹配

函数式编程中还有另一个重要的方面，那就是（结构式）模式匹配。不要将这个概念和正则表达式中的模式匹配相混淆。还记得吗，第1章结束时，我们了解到数学公式可以通过下面的方式进行定义：

```
f(0) = 1
f(n) = n*f(n-1) otherwise
```

不过在Java语言中，你只能通过`if-then-else`语句或者`switch`语句实现。随着数据类型变得愈加复杂，需要处理的代码（以及代码块）的数量也在迅速攀升。使用模式匹配能有效地减少这种混乱的情况。

① 关于延迟计算，可以参考https://wiki.haskell.org/Haskell/Lazy_evaluation。——译者注

为了说明，我们先看一个树结构，你希望能够遍历这一整棵树。我们假设使用一种简单的数学语言，它包含数字和二进制操作符：

```
class Expr { ... }
class Number extends Expr { int val; ... }
class BinOp extends Expr { String opname; Expr left, right; ... }
```

假设你需要编写方法简化一些表达式。比如，`5 + 0`可以简化为5。使用我们的域语言，`new BinOp("+", new Number(5), new Number(0))`可以简化为`Number(5)`。你可以像下面这样遍历`Expr`结构：

```
Expr simplifyExpression(Expr expr) {
    if (expr instanceof BinOp
          && ((BinOp)expr).opname.equals("+"))
          && ((BinOp)expr).right instanceof Number
          && ... // 变得非常笨拙
          && ... ) {
        return (Binop)expr.left;
    }
    ...
}
```

你可以预期这种方式下代码会迅速地变得异常丑陋，难于维护。

14.4.1 访问者设计模式

Java语言中还有另一种方式可以解包数据类型，那就是使用访问者（Visitor）设计模式。本质上，使用这种方法你需要创建一个单独的类，这个类封装了一个算法，可以“访问”某种数据类型。

它是如何工作的呢？访问者类接受某种数据类型的实例作为输入。它可以访问该实例的所有成员。下面是一个例子，通过这个例子我们能了解这一方法是如何工作的。首先，你需要向`BinOp`添加一个`accept`方法，它接受一个`SimplifyExprVisitor`作为参数，并将自身传递给它（你还需要为`Number`添加一个类似的方法）：

```
class BinOp extends Expr{
    ...
    public Expr accept(SimplifyExprVisitor v){
        return v.visit(this);
    }
}
```

`SimplifyExprVisitor`现在就可以访问`BinOp`对象并解包其中的内容了：

14

```
public class SimplifyExprVisitor {
    ...
    public Expr visit(BinOp e){
        if("+".equals(e.opname) && e.right instanceof Number && …){
            return e.left;
        }
```

```
        return e;
    }
}
```

14.4.2 用模式匹配力挽狂澜

通过一个名为模式匹配的特性，我们能以更简单的方案解决问题。这种特性目前在Java语言中暂时还不提供，所以我们会以Scala程序设计语言的一个小例子来展示模式匹配的强大威力。通过这些介绍你能够了解一旦Java语言支持模式匹配，我们能做哪些事情。

假设数据类型Expr代表的是某种数学表达式，在Scala程序设计语言中（我们采用Scala的原因是它的语法与Java非常接近），你可以利用下面的这段代码解析表达式：

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e   // 加0
    case BinOp("*", e, Number(1)) => e   // 乘以1
    case BinOp("/", e, Number(1)) => e   // 除以1
    case _ => expr                       // 不能简化expr
}
```

模式匹配为操纵类树型数据结构提供了一个极其详细又极富表现力的方式。构建编译器或者处理商务规则的引擎时，这一工具尤其有用。注意，Scala的语法

```
Expression match { case Pattern => Expression ... }
```

和Java的语法非常相似：

```
switch (Expression) { case Constant : Statement ... }
```

Scala的通配符判断和Java中的default:扮演这同样的角色。这二者之间主要的语法区别在于Scala是面向表达式的，而Java则更多地面向语句，不过，对程序员而言，它们主要的区别是Java中模式的判断标签被限制在了某些基础类型、枚举类型、封装基础类型的类以及String类型。使用支持模式匹配的语言实践中能带来的最大的好处在于，你可以避免出现大量嵌套的switch或者if-then-else语句和字段选择操作相互交织的情况。

非常明显，Scala的模式匹配在表达的难易程度上比Java更胜一筹，你只能期待未来版本的Java能支持更具表达性的switch语句。我们会在第16章给出更加详细的介绍。

与此同时，让我们看看如何凭借Java 8的Lambda以另一种方式在Java中实现类模式匹配。我们在这里介绍这一技巧的目的仅仅是想让你了解Lambda另一个有趣的应用。

Java中的伪模式匹配

首先，让我们看看Scala的模式匹配特性提供的匹配表达式有多么丰富。比如下面这个例子：

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    ...
```

它表达的意思是：“检查expr是否为BinOp，抽取它的三个组成部分（opname、left、right），紧接着对这些组成部分分别进行模式匹配——第一个部分匹配String+，第二个部分

匹配变量e（它总是匹配），第三个部分匹配模式Number(0)。”换句话说，Scala（以及很多其他的函数式语言）中的模式匹配是多层次的。我们使用Java 8的Lambda表达式进行的模式匹配模拟只会提供一层的模式匹配；以前面的这个例子而言，这意味着它只能覆盖BinOp(op, l, r)或者Number(n)这种用例，无法顾及BinOp("+", e, Number(0))。

首先，我们做一些稍微让人惊讶的观察。由于你选择使用Lambda，原则上你的代码里不应该使用if-then-else。你可以使用方法调用

```
myIf(condition, () -> e1, () -> e2);
```

取代condition ? e1 : e2这样的代码。

在某些地方，比如库文件中，你可能有这样的定义（使用了通用类型T）：

```
static <T> T myIf(boolean b, Supplier<T> truecase, Supplier<T> falsecase) {
    return b ? truecase.get() : falsecase.get();
}
```

类型T扮演了条件表达式中结果类型的角色。原则上，你可以用if-then-else完成类似的事儿。

当然，正常情况下用这种方式会增加代码的复杂度，让它变得愈加晦涩难懂，因为用if-then-else就已经能非常顺畅地完成这一任务，这么做似乎有些杀鸡用牛刀的嫌疑。不过，我们也注意到，Java的switch和if-then-else无法完全实现模式匹配的思想，而Lambda表达式能以简单的方式实现单层的模式匹配——对照使用if-then-else链的解决方案，这种方式要简洁得多。

回来继续讨论类Expr的模式匹配值，Expr类有两个子类，分别为BinOp和Number，你可以定义一个方法patternMatchExpr（同样，我们在这里会使用泛型T，用它表示模式匹配的结果类型）：

```
interface TriFunction<S, T, U, R>{
    R apply(S s, T t, U u);
}

static <T> T patternMatchExpr(
                       Expr e,
                       TriFunction<String, Expr, Expr, T> binopcase,
                       Function<Integer, T> numcase,
                       Supplier<T> defaultcase) {
    return
     (e instanceof BinOp) ?
        binopcase.apply(((BinOp)e).opname, ((BinOp)e).left,
                                           ((BinOp)e).right) :
     (e instanceof Number) ?
        numcase.apply(((Number)e).val) :
        defaultcase.get();
}
```

最终的结果是，方法调用

```
patternMatchExpr(e, (op, l, r) -> {return binopcode;},
                    (n) -> {return numcode;},
                    () -> {return defaultcode;});
```

会判断e是否为BinOp类型（如果是，会执行binopcode方法，它能够通过标识符op、l和r访问BinOp的字段），是否为Number类型（如果是，会执行numcode方法，它可以访问n的值）。这个方法还可以返回defaultcode，如果有人在将来某个时刻创建了一个树节点，它既不是BinOp类型，也不是Number类型，那就会执行这部分代码。

下面这段代码通过简化的加法和乘法表达式展示了如何使用patternMatchExpr。

代码清单14-1　使用模式匹配简化表达式

```
public static Expr simplify(Expr e) {
    TriFunction<String, Expr, Expr, Expr> binopcase =     ← 处理BinOp表达式
        (opname, left, right) -> {
            if ("+".equals(opname)) {                     ← 处理加法
                if (left instanceof Number && ((Number) left).val == 0) {
                    return right;
                }

                if (right instanceof Number && ((Number) right).val == 0) {
                    return left;
                }
            }
            if ("*".equals(opname)) {                     ← 处理乘法
                if (left instanceof Number && ((Number) left).val == 1) {
                    return right;
                }
                if (right instanceof Number && ((Number) right).val == 1) {
                    return left;
                }
            }
            return new BinOp(opname, left, right);
        };
    Function<Integer, Expr> numcase = val -> new Number(val);     ← 处理Number对象
    Supplier<Expr> defaultcase = () -> new Number(0);     ← 如果用户提供的Expr无法识别时进行的默认处理机制

    return patternMatchExpr(e, binopcase, numcase, defaultcase);     ← 进行模式匹配
}
```

你可以通过下面的方式调用简化的方法：

```
Expr e = new BinOp("+", new Number(5), new Number(0));
Expr match = simplify(e);
System.out.println(match);     ← 打印输出5
```

目前为止，你已经学习了很多内容，包括高阶函数、科里化、持久化数据结构、延迟列表以及模式匹配。现在我们看一些更加微妙的技术，为了避免将前面的内容弄得过于复杂，我们刻意地将这部分内容推迟到了后面。

14.5 杂项

这一节里我们会一起探讨两个关于函数式和引用透明性的比较复杂的问题，一个是效率，另一个关乎返回一致的结果。这些都是非常有趣的问题，我们直到现在才讨论它们的原因是它们通常都由副作用引起，并非我们要介绍的核心概念。我们还会探究结合器（Combinator）的思想——即接受两个或多个方法（函数）做参数且返回结果是另一个函数的方法；这一思想直接影响了新增到Java 8中的许多API。

14.5.1 缓存或记忆表

假设你有一个无副作用的方法omputeNumberOfNodes(Range)，它会计算一个树形网络中给定区间内的节点数目。让我们假设，该网络不会发生变化，即该结构是不可变的，然而调用computeNumberOfNodes方法的代价是非常昂贵的，因为该结构需要执行递归遍历。不过，你可能需要多次地计算该结果。如果你能保证引用透明性，那么有一种聪明的方法可以避免这种冗余的开销。解决这一问题的一种比较标准的解决方案是使用记忆表（memoization）——为方法添加一个封装器，在其中加入一块缓存（比如，利用一个HashMap）——封装器被调用时，首先查看缓存，看请求的“(参数，结果)对”是否已经存在于缓存，如果已经存在，那么方法直接返回缓存的结果；否则，你会执行computeNumberOfNodes调用，不过从封装器返回之前，你会将新计算出的“(参数，结果)对”保存到缓存中。严格地说，这种方式并非纯粹的函数式解决方案，因为它会修改由多个调用者共享的数据结构，不过这段代码的封装版本的确是引用透明的。

实际操作上，这段代码的工作如下：

```
final Map<Range,Integer> numberOfNodes = new HashMap<>();
Integer computeNumberOfNodesUsingCache(Range range) {
    Integer result = numberOfNodes.get(range);
    if (result != null){
        return result;
    }
    result = computeNumberOfNodes(range);
    numberOfNodes.put(range, result);
    return result;
}
```

注意 Java 8改进了Map接口，提供了一个名为computeIfAbsent的方法处理这样的情况。我们会在附录B介绍这一方法。但是，我们在这里也提供一些参考，你可以用下面的方式调用computeIfAbsent方法，帮助你编写结构更加清晰的代码：

```
Integer computeNumberOfNodesUsingCache(Range range) {
    return numberOfNodes.computeIfAbsent(range,
                                          this::computeNumberOfNodes);
}
```

14

很明显，方法computeNumberOfNodesUsingCache是引用透明的（我们假设compute-NumberOfNodes也是引用透明的）。不过，事实上，numberOfNodes处于可变共享状态，并且HashMap也没有同步[①]，这意味这该段代码不是线程安全的。如果多个核对numberOfNodes执行并发调用，即便不用HashMap，而是用（由锁保护的）Hashtable或者（并发无锁的）ConcurrentHashMap，可能都无法达到预期的性能，因为这中间又存在由于发现某个值不在Map中，需要将对应的“(参数，结果）对”插回到Map而引起的条件竞争。这意味着多个核上的进程可能算出的结果相同，又都需要将其加入到Map中。

从刚才讨论的各种纠结中，我们能得到的最大收获可能是，一旦并发和可变状态的对象揉到一起，它们引起的复杂度要远超我们的想象，而函数式编程能从根本上解决这一问题。当然，这也有一些例外，比如出于底层性能的优化，可能会使用缓存，而这可能会有一些影响。另一方面，如果不使用缓存这样的技巧，如果你以函数式的方式进行程序设计，那就完全不必担心你的方法是否使用了正确的同步方式，因为你清楚地知道它没有任何共享的可变状态。

14.5.2　“返回同样的对象”意味着什么

让我们在次回顾一下14.2.3节中二叉树的例子。图14-4中，变量t指向了一棵现存的树，依据该图，调用fupdate(fupdate("Will",26, t)会生成一个新的树，这里我们假设该树会被赋值给变量t2。通过该图，我们非常清楚地知道变量t，以及所有它涉及的数据结构都是不会变化的。现在，假设你在新增的赋值操作中执行一次字面上和上一操作完全相同的调用，如下所示：

```
t3 = fupdate("Will", 26, t);
```

这时t会指向第三个新创建的节点，该节点包含了和t2一样的数据。好，问题来了：fupdate是否符合引用透明性原则呢？*引用透明性*原则意味着“使用相同的参数（即这个例子的情况）产生同样的结果”。问题是t2和t3属于不同的对象引用，所以(t2==t3)这一结论并不成立，这样说起来你只能得出一个结论：fupdate并不符合引用透明性原则。虽然如此，使用不会改动的持久化数据结构时，t2和t3在逻辑上并没有差别。 对于这一点我们已经辩论了很长时间，不过最简单的概括可能是函数式编程通常不使用==（引用相等），而是使用equal对数据结构值进行比较，由于数据没有发生变更，所以这种模式下fupdate是引用透明的。

14.5.3　结合器

函数式编程时编写高阶函数是非常普通而且非常自然的事。高阶函数接受两个或多个函数，并返回另一个函数，实现的效果在某种程度上类似于将这些函数进行了结合。术语*结合器*通常用于描述这一思想。Java 8中的很多API都受益于这一思想，比如CompletableFuture类中的

① 这是极其容易滋生缺陷的地方。我们很容易随意地使用HashMap，却忘记了Java文档中的提示，这一数据结构不是线程安全的（或者简单地说，由于我们的程序是单线程的，而毫无顾忌地使用）。

thenCombine方法。该方法接受两个CompletableFuture方法和一个BiFunction方法，返回另一个CompletableFuture方法。

虽然深入探讨函数式编程中结合器的特性已经超出了本书的范畴，了解结合器使用的一些特例还是非常有价值的，它能让我们切身体验函数式编程中构造接受和返回函数的操作是多么普通和自然。下面这个方法就体现了函数组合（function composition）的思想：

```
static <A,B,C> Function<A,C> compose(Function<B,C> g, Function<A,B> f) {
    return x -> g.apply(f.apply(x));
}
```

它接受函数f和g作为参数，并返回一个函数，实现的效果是先做f，接着做g。你可以接着用这种方式定义一个操作，通过结合器完成内部迭代的效果。让我们看这样一个例子，你希望接受一个参数，并使用函数f连续地对它进行操作（比如*n*次），类似循环的效果。我们将你的操作命名为repeat，它接受一个参数f，f代表了一次迭代中进行的操作，它返回的也是一个函数，返回的函数会在*n*次迭代中执行。像下面这样一个方法调用

```
repeat(3, (Integer x) -> 2*x);
```

形成的效果是x ->(2*(2*(2*x)))或者x -> 8*x。

你可以通过下面这段代码进行测试：

```
System.out.println(repeat(3, (Integer x) -> 2*x).apply(10));
```

输出的结果是80。

你可以按照下面的方式编写repeat方法（请特别留意0次循环的特殊情况）：

```
static <A> Function<A,A> repeat(int n, Function<A,A> f) {
    return n==0 ? x -> x
                : compose(f, repeat(n-1, f));
}
```

如果n的值为0，直接返回“什么也不做”的标识符

否则执行函数f，重复执行n-1次，紧接着再执行一次

这个想法稍作变更可以对迭代概念的更丰富外延进行建模，甚至包括对在迭代之间传递可变状态的函数式模型。不过，由于篇幅有限，我们就不再继续展开了，本章的目标只是为大家做一个概率的总结，让大家对Java 8的基石函数式编程有一个全局的观念。市面上还有很多优秀的书籍，对函数式编程进行了更深入的介绍，大家可以选择适合的进一步学习。

14.6 小结

下面是本章中你应该掌握的重要概念。

- 一等函数是可以作为参数传递，可以作为结果返回，同时还能存储在数据结构中的函数。
- 高阶函数接受至少一个或者多个函数作为输入参数，或者返回另一个函数的函数。Java中典型的高阶函数包括comparing、andThen和compose。
- 科里化是一种帮助你模块化函数和重用代码的技术。

- 持久化数据结构在其被修改之前会对自身前一个版本的内容进行备份。因此，使用该技术能避免不必要的防御式复制。
- Java语言中的Stream不是自定义的。
- 延迟列表是Java语言中让Stream更具表现力的一个特性。延迟列表让你可以通过辅助方法（`supplier`）即时地创建列表中的元素，辅助方法能帮忙创建更多的数据结构。
- 模式匹配是一种函数式的特性，它能帮助你解包数据类型。它可以看成Java语言中`switch`语句的一种泛化。
- 遵守“引用透明性”原则的函数，其计算结构可以进行缓存。
- 结合器是一种函数式的思想，它指的是将两个或多个函数或者数据结构进行合并。

第 15 章

面向对象和函数式编程的混合：Java 8和Scala的比较

本章内容

- 什么是Scala语言
- Java 8与Scala是如何相生相承的
- Scala中的函数与Java 8中的函数有哪些区别
- 类和trait

Scala是一种混合了面向对象和函数式编程的语言。它常常被看作Java的一种替代语言，程序员们希望在运行于JVM上的静态类型语言中使用函数式特性，同时又期望保持Java体验的一致性。和Java比较起来，Scala提供了更多的特性，包括更复杂的类型系统、类型推断、模式匹配（我们在14.4节提到过）、定义域语言的结构等。除此之外，你可以在Scala代码中直接使用任何一个Java类库。

你可能会有这样的疑惑，我们为什么要在一本介绍Java 8的书里特别设计一章讨论Scala。本书的绝大部分内容都在介绍如何在Java中应用函数式编程。Scala和Java 8极其类似，它们都支持对集合的函数式处理（类似于对Stream的操作）、一等函数、默认方法。不过Scala将这些思想向前又推进了一大步：它为实现这些思想提供了大量的特性，这方面它领先了Java 8一大截。我们相信你会发现，对比Scala和Java 8在实现方式上的不同以及了解Java 8目前的局限是非常有趣的。通过这一章，我们希望能针对这些问题为你提供一些线索，解答一些疑惑。

请记住，本章的目的并非让你掌握如何编写纯粹的Scala代码，或者了解Scala的方方面面。不少的特性，比如模式匹配，在Scala中是天然支持的，也非常容易理解，不过这些特性在Java 8中却并未提供，这部分内容我们在这里不会涉及。本章着重对比Java 8中新引入的特性和该特性在Scala中的实现，帮助你更全面地理解该特性。比如，你会发现，用Scala重新实现原先用Java完成的代码更简单，可读性也更好。

本章从对Scala的介绍入手：让你了解如何使用Scala编写简单的程序，以及如何处理集合。紧接着我们会讨论Scala中的函数式，包括一等函数、闭包以及科里化。最后，我们会一起看看Scala中的类，以及一种名为trait的特性，它是Scala中带默认方法的接口。

15.1　Scala 简介

本节会简要地介绍Scala的一些基本特性，让你有一个比较直观的感受：到底简单的Scala程序怎么编写。我们从一个略微改动的Hello World示例入手，该程序会以两种方式编写，一种以命令式的风格编写，另一种以函数式的风格编写。接着，我们会看看Scala支持哪些数据结构——List、Set、Map、Stream、Tuple以及Option——并将它们与Java 8中对应的数据结构一一进行比较。最后，我们会介绍trait，它是Scala中接口的替代品，支持在对象实例化时对方法进行继承。

15.1.1　你好，啤酒

让我们看一个简单的例子，这样你能对Scala的语法、语言特性，以及它与Java的差异有一个比较直观的认识。我们对经典的Hello World示例进行了微调，让我们来点儿啤酒。你希望在屏幕上打印输出下面这些内容：

```
Hello 2 bottles of beer
Hello 3 bottles of beer
Hello 4 bottles of beer
Hello 5 bottles of beer
Hello 6 bottles of beer
```

1. 命令式Scala

下面这段代码中，Scala以命令式的风格打印输出这段内容：

```
object Beer {
  def main(args: Array[String]){
    var n : Int = 2
    while( n <= 6 ){
      println(s"Hello ${n} bottles of beer")    // 在字符串中插值
      n += 1
    }
  }
}
```

如何运行这段代码的指导信息可以在Scala的官方网站找到[①]。这段代码看起来和你用Java编写的程序相当类似。它的结构和Java程序几乎一样：它包含了一个名为main的方法，该方法接受一个由参数构成的数组（类型注释遵循这样的语法s : String，不像Java那样用String s）。由于main方法不返回值，所以使用Scala不需要像Java那样声明一个类型为void的返回值。

注意　通常而言，在Scala中声明非递归的方法时，不需要显式地返回类型，因为Scala会自动地替你推断生成一个。

转入main的方法体之前，我们想先讨论下对象的声明。不管怎样，Java中的main方法都需

① 参见http://www.scala-lang.org/documentation/getting-started.html。

要在某个类中声明。对象的声明产生了一个单例的对象：它声明了一个对象，比如Bear，与此同时又对其进行了实例化。整个过程中只有一个实例被创建。这是第一个以经典的设计模式（即单例模式）实现语言特性的例子——尽量不拘一格地使用它！此外，你可以将对象声明中的方法看成静态的，这也是main方法的方法签名中并未显式地声明为静态的原因。

现在让我们看看main的方法体。它看起来和Java非常类似，但是语句不需要再以分号结尾了（它成了一种可选项）。方法体中包含了一个while循环，它会递增一个可修改变量n。通过预定义的方法println，你可以打印输出n的每一个新值。println这一行还展示了Scala的另一个特性：字符串插值。字符串插值在字符串的字面量中内嵌变量和表达式。前面的这段代码中，你在字符串字面量s"Hello ${n} bottles of beer"中直接使用了变量n。字符串前附加的插值操作符s，神奇地完成了这一转变。而在Java中，你通常需要使用显式的连接操作，比如"Hello " + n + " bottles of beer"，才能达到同样的效果。

2. 函数式Scala

那么，Scala到底能带来哪些好处呢？毕竟我们在本书里主要讨论的还是函数式。前面的这段代码利用Java 8的新特性能以更加函数式的方式实现，如下所示：

```
public class Foo {
    public static void main(String[] args) {
        IntStream.rangeClosed(2, 6)
                 .forEach(n -> System.out.println("Hello " + n +
                                                  " bottles of beer"));
    }
}
```

如果以Scala来实现，它是下面这样的：

```
object Beer {
  def main(args: Array[String]){
    2 to 6 foreach { n => println(s"Hello ${n} bottles of beer") }
  }
}
```

这种实现看起来和基于Java的版本有几分相似，不过Scala的实现更加简洁。首先，你使用表达式2 to 6创建了一个区间。这看起来相当特别： 2在这里并非原始数据类型，在Scala中它是一个类型为Int的对象。Scala语言里，任何事物都是对象；不像Java那样，Scala没有原始数据类型一说了。通过这种方式，Scala被转变成为了纯粹的面向对象语言。Scala语言中Int对象支持名为to的方法，它接受另一个Int对象，返回一个区间。所以，你还可以通过另一种方式实现这一语句，即2.to(6)。由于接受一个参数的方法可以采用中缀式表达，所以你可以用开头的方式实现这一语句。紧接着，我们看到了foreach（这里的e采用的是小写），它和Java 8中的forEach（使用了大写的E）也很类似。它是对一个区间进行操作的函数（这里你可以再次使用中缀表达式），它可以接受Lambda表达式做参数，对区间的每一个元素顺次执行操作。这里Lambda表达式的语法和Java 8也非常类似，区别是箭头的表示用=>替换了->[①]。前面的这段代码是函数式的：因为

① 注意，在Scala语言中，我们使用“匿名函数”或者“闭包”（可以互相替换）来指代Java 8中的Lambda表达式。

就像我们早期使用while循环时示例的那样，你并未修改任何变量。

15.1.2　基础数据结构：List、Set、Map、Tuple、Stream 以及 Option

几杯啤酒之后，你一定已经止住口渴，精神一振了吧？大多数的程序都需要操纵和存储数据，那么，就让我们一起看看如何在Scala中操作集合，以及它与Java 8中操作的不同。

1. 创建集合

在Scala中创建集合是非常简单的，这主要归功于它对简洁性的一贯坚持。比如，创建一个Map，你可以用下面的方式：

```
val authorsToAge = Map("Raoul" -> 23, "Mario" -> 40, "Alan" -> 53)
```

这行代码中，有几件事情是我们首次碰到的。首先，你使用->语法轻而易举地创建了一个Map，并完成了键到值的映射，整个过程令人吃惊地简单。你不再需要像Java中那样手工添加每一个元素：

```
Map<String, Integer> authorsToAge = new HashMap<>();
authorsToAge.put("Raoul", 23);
authorsToAge.put("Mario", 40);
authorsToAge.put("Alan", 53);
```

关于这一点，也有一些讨论，希望在未来的Java版本中添加类似的语法糖，不过在Java 8[①]中暂时还没有这样的特性。第二件让人耳目一新的事是你可以选择不对变量authorsToAge的类型进行注解。实际上，你可以编写val authorsToAge : Map[String, Int]这样的代码，显式地声明变量类型，不过Scala可以替你推断变量的类型（请注意，即便如此，代码依旧是静态检查的！所有的变量在编译时都具有确定的类型）。我们会在本章后续部分继续讨论这一特性。第三，你可以使用val关键字替换var。这二者之间存在什么差别吗？关键字val表明变量是只读的，并由此不能被赋值（就像Java中声明为final的变量一样）。而关键字var表明变量是可以读写的。

听起来不错，那么其他的集合类型呢？你可以用同样的方式轻松地创建List（一种单向链表）或者Set（不带冗余数据的集合），如下所示：

```
val authors = List("Raoul", "Mario", "Alan")
val numbers = Set(1, 1, 2, 3, 5, 8)
```

这里的变量authors包含3个元素，而变量numbers包含5个元素。

2. 不可变与可变的比较

Scala的集合有一个重要的特质我们应该牢记在心，那就是我们之前创建的集合在默认情况下都是只读的。这意味着它们从创建开始就不能修改。这是一种非常有用的特性，因为有了它，你知道任何时候访问程序中的集合都会返回包含相同元素的集合。

那么，你怎样才能更新Scala语言中不可变的集合呢？回到前面章节介绍的术语，Scala中的这些集合都是持久化的：更新一个Scala集合会生成一个新的集合，这个新的集合和之前版本的集

① 参见http://openjdk.java.net/jeps/186。

合共享大部分的内容，最终的结果是数据尽可能地实现了持久化，避免了图14-3和图14-4中那样由于改变所引起的问题。由于具备这一属性，你代码的隐式数据依赖更少： 对你代码中集合变更的困惑（比如在何处更新了集合，什么时候做的更新）也会更少。

让我们看一个实际的例子，具体分析下这一思想是如何影响你的程序设计的。下面这段代码中，我们会为Set添加一个元素：

```
val numbers = Set(2, 5, 3);
val newNumbers = numbers + 8
println(newNumbers)
println(numbers)
```

这里的操作符+会将8添加到Set中，创建并返回一个新的Set对象

(2, 5, 3, 8)

(2, 5, 3)

这个例子中，原始Set对象中的数字没有发生变更。实际的效果是该操作创建了一个新的Set，并向其中加入了一个新的元素。

注意，Scala语言并未强制你必须使用不可变集合，它只是让你能更轻松地在你的代码中应用不可变原则。scala.collection.mutable包中也包含了集合的可变版本。

不可修改与不可变的比较

Java中提供了多种方法创建**不可修改的**（unmodifiable）集合。下面的代码中，变量newNumbers是集合Set对象numbers的一个只读视图：

```
Set<Integer> numbers = new HashSet<>();
Set<Integer> newNumbers = Collections.unmodifiableSet(numbers);
```

这意味着你无法通过操作变量newNumbers向其中加入新的元素。不过，不可修改集合仅仅是对可变集合进行了一层封装。通过直接访问numbers变量，你还是能向其中加入元素。

与此相反，**不可变**（immutable）集合确保了该集合在任何时候都不会发生变化，无论有多少个变量同时指向它。

我们在第14章介绍过如何创建一个持久化的数据结构：你需要创建一个不可变数据结构，该数据结构会保存它自身修改之前的版本。任何的修改都会创建一个更新的数据结构。

3. 使用集合

现在你已经了解了如何创建结合，你还需要了解如何使用这些集合开展工作。我们很快会看到Scala支持的集合操作和Stream API提供的操作极其类似。比如，在下面的代码片段中，你会发现熟悉的filter和map，图15-1对这段代码逻辑进行了阐释。

```
val fileLines = Source.fromFile("data.txt").getLines.toList()
val linesLongUpper
  = fileLines.filter(l => l.length() > 10)
             .map(l => l.toUpperCase())
```

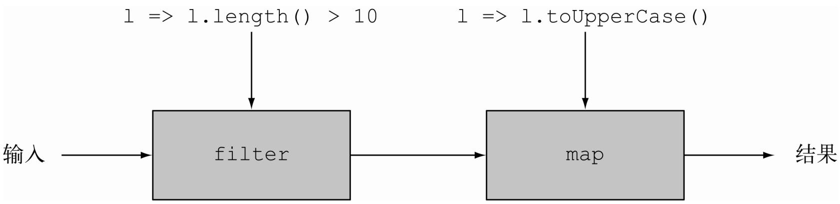


图15-1　使用Scala的`List`实现类`Stream`操作

不用担心第一行的内容，它实现的基本功能是将文件中的所有行转换为一个字符串列表（类似Java 8提供的`Files.readAllLines`）。第二行创建了一个由两个操作构成的流水线：

- `filter`操作会过滤出所有长度超过10的行
- `map`操作会将这些长的字符串统一转换为大写字符

这段代码也可以用下面的方式实现：

```
val linesLongUpper
  = fileLines filter (_.length() > 10) map(_.toUpperCase())
```

这段代码使用了中缀表达式和下划线（_），下划线是一种占位符，它按照位置匹配对应的参数。这个例子中，你可以将`_.length()`解读为`l =>l.length()`。在传递给`filter`和`map`的函数中，下划线会被绑定到待处理的`line`参数。

Scala的集合API提供了很多非常有用的操作。我们强烈建议你抽空浏览一下Scala的文档，对这些API有一个大致的了解[①]。注意，Scala的集合类提供的功能比Stream API提供的功能还丰富很多，比如，Scala的集合类支持压缩操作，你可以将两个列表中的元素整合到一个列表中。通过学习，一定能大大增强你的功力。这些编程技巧在将来的Java版本中也可能会被Stream API所引入。

最后，还记得吗？Java 8中你可以对`Stream`调用`parallel`方法，将流水线转化为并行执行。Scala提供了类似的技巧；你只需要使用方法`par`就能实现同样的效果：

```
val linesLongUpper
  = fileLines.par filter (_.length() > 10) map(_.toUpperCase())
```

4. 元组

现在，让我们看看另一个特性，该特性使用起来通常异常繁琐，它就是元组。你可能希望使用元组将人的名字和电话号码组合起来，同时又不希望额外声明新的类，并对其进行实例化。你希望元组的结构就像：(“Raoul”，“+ 44 007007007”)、(“Alan”，“+44 003133700”)，诸如此类。

非常不幸，Java目前还不支持元组，所以你只能创建自己的数据结构。下面是一个简单的`Pair`类定义：

```
public class Pair<X, Y> {
    public final X x;
```

① www.scala-lang.org/api/current/#package中既包含了著名的包，也包含一些不那么有名的包的介绍。

```
    public final Y y;
    public Pair(X x, Y y){
        this.x = x;
        this.y = y;
    }
}
```

当然，你还需要显式地实例化Pair对象：

```
Pair<String, String> raoul = new Pair<>("Raoul", "+ 44 007007007");
Pair<String, String> alan = new Pair<>("Alan", "+44 003133700");
```

好了，看起来一切顺利，不过如果是三元组呢？如果是自定义大小的元组呢？这个问题就变得相当繁琐，最终会影响你代码的可读性和可维护性。

Scala提供了名为*元组字面量*的特性来解决这一问题，这意味着你可以通过简单的语法糖创建元组，就像普通的数学符号那样：

```
val raoul = ("Raoul", "+ 44 887007007")
val alan = ("Alan", "+44 883133700")
```

Scala支持任意大小[①]的元组，所以下面的这些声明都是合法的：

```
val book = (2014, "Java 8 in Action", "Manning")    // 元组类型为(Int, String, String)
val numbers = (42, 1337, 0, 3, 14)    // 元组类型为(Int, Int, Int, Int, Int)
```

你可以依据它们的位置，通过存取器（accessor）_1、_2（从1开始的一个序列）访问元组中的元素，比如：

```
println(book._1)    // 打印输出2014
println(numbers._4)    // 打印输出3
```

是不是比Java语言中现有的实现方法简单很多？好消息是关于将元组字面量引入到未来Java版本的讨论正在进行中（我们会在第16章围绕这一主题进行更深入的讨论）。

5. Stream

到目前为止，我们讨论的集合，包括List、Set、Map和Tuple都是即时计算的（即在第一时间立刻进行计算）。当然，你也已经了解Java 8中的Stream是按需计算的（即延迟计算）。通过第5章，你知道由于这一特性，Stream可以表示无限的序列，同时又不消耗太多的内存。

Scala也提供了对应的数据结构，它采用延迟方式计算数据结构，名称也叫Stream！不过Scala中的Stream提供了更加丰富的功能，让Java中的Stream有些黯然失色。Scala中的Stream可以记录它曾经计算出的值，所以之前的元素可以随时进行访问。除此之外，Stream还进行了索引，所以Stream中的元素可以像List那样通过索引访问。注意，这种抉择也附带着开销，由于需要存储这些额外的属性，和Java 8中的Stream比起来，Scala版本的Stream内存的使用效率变低了，因为Scala中的Stream需要能够回溯之前的元素，这意味着之前访问过的元素都需要在内存"记录下来"（即进行缓存）。

① 元组中元素的最大上限为23。

6. Option

另一个你熟悉的数据结构是Option。我们在第10章中讨论过Java的Optional，Option是Java 8中Optional类型的Scala版本。我们建议你在设计API时尽可能地使用Optional，这种方式下，接口用户只需要阅读方法签名就能了解他们是否应该传递一个optional值。我们应该尽量地用它替代null，避免发生空指针异常。

第10章中，你了解了我们可以使用Optional返回客户的保险公司名称——如果客户的年龄超过设置的最低值，就返回该客户对应的保险公司名称，具体代码如下：

```
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

在Scala语言中，你可以使用Option使用Optional类似的方法实现该函数：

```
def getCarInsuranceName(person: Option[Person], minAge: Int) =
  person.filter(_.getAge() >= minAge)
        .flatMap(_.getCar)
        .flatMap(_.getInsurance)
        .map(_.getName).getOrElse("Unknown")
```

这段代码中除了getOrElse方法，其他的结构和方法你一定都非常熟悉，getOrElse是与Java 8中orElse等价的方法。你看到了吗？在本书中学习的新概念能直接应用于其他语言！然而，不幸的是，为了保持同Java的兼容性，在Scala中依旧保持了null，不过我们极度不推荐你使用它。

> **注意**　在前面的代码中，你使用的是_.getCar（并未使用圆括号），而不是_.getCar()（带圆括号）。Scala语言中，执行方法调用时，如果不需要传递参数，那么函数的圆括号是可以省略的。

15.2　函数

Scala中的函数可以看成为了完成某个任务而组合在一起的指令序列。它们对于抽象行为非常有帮助，是函数式编程的基石。

对于Java语言中的方法，你已经非常熟悉了：它们是与类相关的函数。你也已经了解了Lambda表达式，它可以看成一种匿名函数。跟Java比较起来，Scala为函数提供的特性要丰富得多，我们在这一节中会逐一讲解。Scala提供了下面这些特性。

- 函数类型，它是一种语法糖，体现了Java语言中函数描述符的思想，即，它是一种符号，表示了在函数接口中声明的抽象方法的签名。这些内容我们在第3章中都介绍过。
- 能够读写非本地变量的匿名函数，而Java中的Lambda表达式无法对非本地变量进行写

操作。

- 对科里化的支持，这意味着你可以将一个接受多个参数的函数拆分成一系列接受部分参数的函数。

15.2.1 Scala 中的一等函数

函数在Scala语言中是一等值。这意味着它们可以像其他的值，比如Integer或者String那样，作为参数传递，可以作为结果值返回。正如我们在前面章节所介绍的那样，Java 8中的方法引用和Lambda表达式也可以看成一等函数。

让我们看一个例子，看看Scala中的一等函数是如何工作的。我们假设你现在有一个字符串列表，列表中的值是朋友们发送给你的消息（tweet）。你希望依据不同的筛选条件对该列表进行过滤，比如，你可能想要找出所有提及Java这个词或者短于某个长度的消息。你可以使用谓词（返回一个布尔型结果的函数）定义这两个筛选条件，代码如下：

```
def isJavaMentioned(tweet: String) : Boolean = tweet.contains("Java")

def isShortTweet(tweet: String) : Boolean = tweet.length() < 20
```

Scala语言中，你可以直接传递这两个方法给内嵌的filter，如下所示（这和你在Java中使用方法引用将它们传递给某个函数大同小异）：

```
val tweets = List(
    "I love the new features in Java 8",
    "How's it going?",
    "An SQL query walks into a bar, sees two tables and says 'Can I join you?'"
)

tweets.filter(isJavaMentioned).foreach(println)
tweets.filter(isShortTweet).foreach(println)
```

现在，让我们一起审视下内嵌方法filter的函数签名：

```
def filter[T](p: (T) => Boolean): List[T]
```

你可能会疑惑参数p到底代表的是什么类型（即(T) => Boolean），因为在Java语言里你期望看到的是一个函数接口！这其实是一种新的语法，Java中暂时还不支持。它描述的是一个*函数类型*。这里它表示的是这样一个函数，它接受类型为T的对象，返回一个布尔类型的值。Java语言中，它被编码为Predicate<T>或者Function<T, Boolean>。所以它实际上和isJavaMentioned和isShortTweet具有类似的函数签名，所以你可以将它们作为参数传递给filter方法。Java 8语言的设计者们为了保持语言与之前版本的一致性，决定不引入类似的语法。对于一门语言的新版本，引入太多的新语法会增加它的学习成本，带来额外学习负担。

15.2.2 匿名函数和闭包

Scala也支持匿名函数。匿名函数和Lambda表达式的语法非常类似。下面的这个例子中，你

将一个匿名函数赋值给了名为isLongTweet的变量，该匿名函数的功能是检查给定的消息长度，判断它是否超长：

```
val isLongTweet : String => Boolean    <-- 这是一个函数类型的变量，它接受一个String参数，返回一个布尔类型的值
    = (tweet : String) => tweet.length() > 60    <-- 一个匿名函数
```

在新版的Java中，你可以使用Lambda表达式创建函数式接口的实例。Scala也提供了类似的机制。前面的这段代码是Scala中声明匿名类的语法糖。Function1（只带一个参数的函数）提供了apply方法的实现：

```
val isLongTweet : String => Boolean
    = new Function1[String, Boolean] {
        def apply(tweet: String): Boolean = tweet.length() > 60
      }
```

由于变量isLongTweet中保存了类型为Function1的对象，你可以调用它的apply方法，这看起来就像下面的方法调用：

```
isLongTweet.apply("A very short tweet")    <-- 返回false
```

如果用Java，你可以采用下面的方式：

```
Function<String, Boolean> isLongTweet = (String s) -> s.length() > 60;
boolean long = isLongTweet.apply("A very short tweet");
```

为了使用Lambda表达式，Java提供了几种内置的函数式接口，比如Predicate、Function、Consumer。Scala提供了trait（你可以暂时将trait想象成接口，我们会在接下来的一节介绍它们）来实现同样的功能：从Function0（一个函数不接受任何参数，并返回一个结果）到Function22（一个函数接受22个参数），它们都定义了apply方法。

Scala还提供了另一个非常酷炫的特性，你可以使用语法糖调用apply方法，效果就像一次函数调用：

```
isLongTweet("A very short tweet")    <-- 返回false
```

编译器会自动地将方法调用f(a)转换为f.apply(a)。更一般地说，如果f是一个支持apply方法的对象（注，apply可以有任意数目的参数），对方法f(a1, ..., an)的调用会被转换为f.apply(a1, ..., an)。

闭包

第3章中我们曾经抛给大家一个问题：Java中的Lambda表达式是否是借由闭包组成的。温习一下，那么什么是闭包呢？闭包是一个函数实例，它可以不受限制地访问该函数的非本地变量。不过Java 8中的Lambda表达式自身带有一定的限制：它们不能修改定义Lambda表达式的函数中的本地变量值。这些变量必须隐式地声明为final。这些背景知识有助于我们理解“Lambda避免了对变量值的修改，而不是对变量的访问”。

与此相反，Scala中的匿名函数可以取得自身的变量，但并非变量当前指向的变量值。比如，下面这段代码在Scala中是可能的：

```
def main(args: Array[String]) {
    var count = 0
    val inc = () => count+=1        <— 这是一个闭包，它捕获并递增count
    inc()
    println(count)               <— 打印输出1
    inc()
    println(count)      <— 打印输出2
}
```

不过在Java中，下面的这段代码会遭遇编译错误，因为count隐式地被强制定义为final：

```
public static void main(String[] args) {
    int count = 0;
    Runnable inc = () -> count+=1;        <— 错误：count必须为final或者在效果上为final
    inc.run();
    System.out.println(count);
    inc.run();
}
```

我们在第7、13以及14章多次提到你应该尽量避免修改，这样你的代码更加易于维护和并发运行，所以请在绝对必要时才使用这一特性。

15.2.3 科里化

第14章中，我们描述了一种名为科里化的技术：带有两个参数（比如x和y）的函数f可以看成一个仅接受一个参数的函数g，函数g的返回值也是一个仅带一个参数的函数。这一定义可以归纳为接受多个参数的函数可以转换为多个接受一个参数的函数。换句话说，你可以将一个接受多个参数的函数切分为一系列接受该参数列表子集的函数。Scala为此特别提供了一个构造器，帮助你更加轻松地科里化一个现存的方法。

为了理解Scala到底带来了哪些变化，让我们先回顾一个Java的示例。你定义了一个简单的函数对两个正整数做乘法运算：

```
static int multiply(int x, int y) {
  return x * y;
}
int r = multiply(2, 10);
```

不过这种定义方式要求向其传递所有的参数才能开始工作。你可以人工地对multiple方法进行切分，让其返回另一个函数：

```
static Function<Integer, Integer> multiplyCurry(int x) {
    return (Integer y) -> x * y;
}
```

由multiplyCurry返回的函数会捕获x的值，并将其与它的参数y相乘，然后返回一个整型结果。这意味着你可以像下面这样在一个map中使用multiplyCurry，对每一个元素值乘以2：

15

```
Stream.of(1, 3, 5, 7)
      .map(multiplyCurry(2))
      .forEach(System.out::println);
```

这样就能得到计算的结果2、6、10、14。这种方式工作的原因是map期望的参数为一个函数，而multiplyCurry的返回结果就是一个函数。

现在的Java语言中，为了构造科里化的形式需要你手工地切分函数（尤其是函数有非常多的参数时），这是极其枯燥的事情。Scala提供了一种特殊的语法可以自动完成这部分工作。比如，正常情况下，你定义的multiply方法如下所示：

```
def multiply(x : Int, y: Int) = x * y

val r = multiply(2, 10);
```

该函数的科里化版本如下：

```
def multiplyCurry(x :Int)(y : Int) = x * y      <-- 定义一个科里化函数

val r = multiplyCurry(2)(10)                    <-- 调用该科里化函数
```

使用语法(x: Int)(y: Int)，方法multiplyCurry接受两个由一个Int参数构成的参数列表。与此相反，multiply接受一个由两个Int参数构成的参数列表。当你调用multiplyCurry时会发生什么呢？multiplyCurry的第一次调用使用了单一整型参数（参数x），即multiplyCurry(2)，返回另一个函数，该函数接受参数y，并将其与它捕获的变量x（这里的值为2）相乘。正如我们在14.1.2节介绍的，我们称这个函数是*部分应用的*，因为它并未提供所有的参数。第二次调用对x和y进行了乘法运算。这意味着你可以将对multiplyCurry的第一次调用保存到一个变量中，进行复用：

```
val multiplyByTwo : Int => Int = multiplyCurry(2)
val r = multiplyByTwo(10)                       <-- 20
```

和Java比较起来，在Scala中你不再需要像这里这样手工地提供函数的科里化形式。Scala提供了一种方便的函数定义语法，能轻松地表示函数使用了多个科里化的参数列表。

15.3　类和 trait

现在我们看看类与接口在Java和Scala中的不同。这两种结构在我们设计应用时都很常用。你会看到相对于Java的类和接口，Scala的类和接口提供了更多的灵活性。

15.3.1　更加简洁的 Scala 类

由于Scala也是一门完全的面向对象语言，你可以创建类，并将其实例化生成对象。最基础的形态上，声明和实例化类的语法与Java非常类似。比如，下面是一个声明Hello类的例子：

```
class Hello {
  def sayThankYou(){
    println("Thanks for reading our book")
  }
}
val h = new Hello()
h.sayThankYou()
```

getter方法和setter方法

一旦你定义的类具有了字段，这件事情就变得有意思了。你碰到过单纯只定义字段列表的Java类吗？很明显，你还需要声明一长串的getter方法、setter方法，以及恰当的构造器。多麻烦啊！除此之外，你还需要为每一个方法编写测试。在企业Java应用中，大量的代码都消耗在了这样的类中。比如下面这个简单的Student类：

```
public class Student {

    private String name;
    private int id;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

你需要手工定义构造器对所有的字段进行初始化，还要实现2个getter方法、2个setter方法。一个非常简单的类现在需要超过20行的代码才能实现！有的集成开发环境或者工具能帮你自动生成这些代码，不过你的代码库中还是需要增加大量额外的代码，而这些代码与你实际的业务逻辑并没有太大的关系。

Scala语言中构造器、getter方法以及setter方法都能隐式地生成，从而大大降低你代码中的冗余：

```
class Student(var name: String, var id: Int)    <-- 初始化Student对象
val s = new Student("Raoul", 1)
println(s.name)                                  <-- 取得名称，打印输出Raoul
s.id = 1337                                      <-- 设置id
println(s.id)                                    <-- 打印输出1337
```

15.3.2 Scala 的 trait 与 Java 8 的接口对比

Scala还提供了另一个非常有助于抽象对象的特性，名称叫trait。它是Scala为实现Java中的接口而设计的替代品。trait中既可以定义抽象方法，也可以定义带有默认实现的方法。trait同时还

支持Java中接口那样的多继承，所以你可以将它们看成与Java 8中接口类似的特性，它们都支持默认方法。trait中还可以包含像抽象类这样的字段，而Java 8的接口不支持这样的特性。那么，trait就类似于抽象类吗？显然不是，因为trait支持多继承，而抽象类不支持多继承。Java支持类型的多继承，因为一个类可以实现多个接口。现在，Java 8通过默认方法又引入了对行为的多继承，不过它依旧不支持对状态的多继承，而这恰恰是trait支持的。

为了展示Scala中的trait到底是什么样，让我们看一个例子。我们定义了一个名为`Sized`的trait，它包含一个名为`size`的可变字段，以及一个带有默认实现的`isEmpty`方法：

```
trait Sized{
  var size : Int = 0              <-- 名为size的字段
  def isEmpty() = size == 0       <-- 带默认实现的isEmpty方法
}
```

你现在可以使用一个类在声明时构造它，下面这个例子中`Empty`类的`size`恒定为`0`：

```
class Empty extends Sized           <-- 一个继承自trait Sized的类

println(new Empty().isEmpty())      <-- 打印输出true
```

有一件事非常有趣，trait和Java的接口类似，也是在对象实例化时被创建（不过这依旧是一个编译时的操作）。比如，你可以创建一个`Box`类，动态地决定到底选择哪一个实例支持由trait `Sized`定义的操作：

```
class Box
val b1 = new Box() with Sized       <-- 在对象实例化时构建trait
println(b1.isEmpty())               <-- 打印输出true
val b2 = new Box()
b2.isEmpty()                        <-- 编译错误：因为Box类的声明并未继承Sized
```

如果一个类继承了多个trait，各trait中声明的方法又使用了相同的签名或者相同的字段，这时会发生什么情况？为了解决这些问题，Scala中定义了一系列限制，这些限制和我们之前在第9章介绍默认方法时的限制极其类似。

15.4　小结

下面是这一章中介绍的关键概念和你应该掌握的要点。

- Java 8和Scala都是整合了面向对象编程和函数式编程特性的编程语言，它们都运行于JVM之上，在很多时候可以相互操作。
- Scala支持对集合的抽象，支持处理的对象包括`List`、`Set`、`Map`、`Stream`、`Option`，这些和Java 8非常类似。不过，除此之外Scala还支持元组。
- Scala为函数提供了更加丰富的特性，这方面比Java 8做得好，Scala支持：函数类型、可以不受限制地访问本地变量的闭包，以及内置的科里化表单。
- Scala中的类可以提供隐式的构造器、getter方法以及setter方法。
- Scala还支持trait，它是一种同时包含了字段和默认方法的接口。

第 16 章

结论以及Java的未来

16

本章内容

- Java 8的新特性以及其对编程风格颠覆性的影响
- 由Java 8萌生的一些尚未成熟的编程思想
- Java 9以及Java 10可能发生的变化

我们在本书中讨论了很多内容，希望你现在已经有足够的信心开始使用Java 8编写你自己的代码，或者编译书中提供的例子和测验。这一章里，我们会回顾我们的Java 8学习之路和函数式编程这一潮流。除此之外，还会展望在Java 8之后的版本中可能出现的新的改进和重大的新特性。

16.1 回顾 Java 8 的语言特性

Java 8是一种实践性强、实用性好的语言，想要很好地理解它，方法之一是重温它的各种特性。本章不会简单地罗列Java 8的各种特性，而是会将这些特性串接起来，希望大家不仅能理解这些新特性，还能从语言设计的高度理解Java 8中语言设计的连贯性。作为回顾，本章的另一个目的是阐释Java 8的这些新特性是如何促进Java函数式编程风格的发展的。请记住，这些新特性并非语言设计上的突发奇想，而是一种刻意的设计，它源于两种趋势，即我们在第1章中所说的形势的变化。

- 对多核处理器处理能力的需求日益增长，虽然硅开发技术也在不断进步，但依据摩尔定律每年新增的晶体管数量已经无法使独立CPU核的速度更快了。简单来说，要让你的代码运行得更快，需要你的代码具备并行运算的能力。
- 更简洁地调度以显示风格处理数据的数据集合，这一趋势不断增长。比如，创建一些数据源，抽象所有数据以符合给定的标准，给结果运用一些操作，而不是概括结果或者将结果组成集合以后再做进一步处理。这一风格与使用不变对象和集合相关，它们之后会进一步生成不变值。

不过这两种诉求都不能很好地得到传统的、面向对象编程的支持，命令式的方式和通过迭代器访问修改字段都不能满足新的需要。在CPU的一个核上修改数据，在另一个核上读取该数据的

值，这种方式的代价是非常高的，更不用说你还需要考虑容易出错的锁；类似地，当你的思考局限于通过迭代访问和修改现存的对象时，类流（stream-like）式编程方法看起来就非常地异类。不过，这两种新的潮流都能通过使用函数式编程非常轻松地得到支持，这也解释了为什么Java 8的重心要从我们最初理解的Java大幅地转型。

现在，我们一起从统一、宏观的角度来回顾一下，看看我们都从这本书中学习了哪些东西，它们又是如何相互协作构建出一片新的编程天地的。

16.1.1　行为参数化（Lambda 以及方法引用）

为了编写可重用的方法，比如`filter`，你需要为其指定一个参数，它能够精确地描述过滤条件。虽然Java专家们使用之前的版本也能达到同样的目的（将过滤条件封装成类的一个方法，传递该类的一个实例），但这种方案却很难推广，因为它通常非常臃肿，既难于编写，也不易于维护。

正如你在第2章和第3章中所了解的，Java 8通过借鉴函数式编程，提供了一种新的方式——通过向方法传递代码片段来解决这一问题。这种新的方法非常方便地提供了两种变体。

- 传递一个Lambda表达式，即一段精简的代码片段，比如

```
apple -> apple.getWeight() > 150
```

- 传递一个*方法引用*，该方法引用指向了一个现有的方法，比如这样的代码：

```
Apple::isHeavy
```

这些值具有类似`Function<T, R>`、`Predicate<T>`或者`BiFunction<T, U, R>`这样的类型，值的接收方可以通过`apply`、`test`或其他类似的方法执行这些方法。Lambda表达式自身是一个相当酷炫的概念，不过Java 8对它们的使用方式——将它们与全新的Stream API相结合，最终把它们推向了新一代Java的核心。

16.1.2　流

集合类、迭代器，以及`for-each`结构在Java中历史悠久，也为广大程序员所熟知。直接在集合类中添加`filter`或者`map`这样的方法，利用我们前面介绍的Lambda实现类数据库查询对于Java 8的设计者而言要简单得多。不过他们并没有采用这种方式，而是引入了一套全新的Stream API，即第4章到第7章所介绍的内容——这是值得我们深思的，他们为什么要这么做呢？

集合到底有什么问题，以至于我们需要另起炉灶替换掉它们，或通过一个类似却不同的概念Stream对其进行增强。我们把二者之间的差异概括如下：如果你有一个数据量庞大的集合，你需要对这个集合应用三个操作，比如对这个集合中的对象进行映射，对其中的两个字段进行求和，这之后依据某种条件过滤出满足条件的和，最后对结果进行排序，即为得到结果你需要分三次遍历集合。Stream API则与之相反，它采用延迟算法将这些操作组成一个流水线，通过单次流遍历，一次性完成所有的操作。对于大型的数据集，这种操作方式要高效得多。不过，还有一些需要我们考虑的因素，比如内存缓存，数据集越大，越需要尽可能地减少遍历的次数。

还有其他一些原因也会影响元素并发处理的能力，这些也非常关键，对高效地利用多处理器的能力至关重要。Stream，尤其是它的parallel方法能帮助将一个Stream标记为适合进行并行处理。还记得吗？并行处理和对象的可变状态是水火不容的，所以核心的函数式概念（如我们在第4章中介绍的，包括无副作用的操作，通过Lambda表达式和方法引用对方法进行参数化，用内部迭代替换外部迭代）对于并行使用map、filter或者其他方法发掘Stream的处理能力非常重要。

现在，让我们看看这些观念（介绍Stream时使用过这些术语）怎样直接影响了Completable-Future类的设计。

16.1.3 CompletableFuture

Java从Java 5版本就提供了Future接口。Future对于充分利用多核处理能力是非常有益的，因为它允许一个任务在一个新的核上生成一个新的子线程，新生成的任务可以和原来的任务同时运行。原来的任务需要结果时，它可以通过get方法等待Future运行结束（生成其计算的结果值）。

第11章介绍了Java 8中对Future的CompletableFuture实现。这里再次利用了Lambda表达式。一个非常有用，不过不那么精确的格言这么说："Completable-Future对于Future的意义就像Stream之于Collection。"让我们比较一下这二者。

- 通过Stream你可以对一系列的操作进行流水线，通过map、filter或者其他类似的方法提供行为参数化，它可有效避免使用迭代器时总是出现模板代码。
- 类似地，CompletableFuture提供了像thenCompose、thenCombine、allOf这样的操作，对Future涉及的通用设计模式提供了函数式编程的细粒度控制，有助于避免使用命令式编程的模板代码。

这种类型的操作，虽然大多数只能用于非常简单的场景，不过仍然适用于Java 8的Optional操作，我们一起来回顾下这部分内容。

16.1.4 Optional

Java 8的库提供了Optional<T>类，这个类允许你在代码中指定哪一个变量的值既可能是类型T的值，也可能是由静态方法Optional.empty表示的缺失值。无论是对于理解程序逻辑，抑或是对于编写产品文档而言，这都是一个重大的好消息，你现在可以通过一种数据类型表示显式缺失的值——使用空指针的问题在于你无法确切了解出现空指针的原因，它是预期的情况，还是说由于之前的某一次计算出错导致的一个偶然性的空值，有了Optional之后你就不需要再使用之前容易出错的空指针来表示缺失的值了。

正如我们在第10章中讨论的，如果在程序中始终如一地使用Optional<T>，你的应用应该永远不会发生NullPointerException异常。你可以将这看成另一个绝无仅有的特性，它和Java 8中其他部分都不直接相关，问自己一个问题："为什么用一种表示值缺失的形式替换另一种能帮助我们更好地编写程序？"进一步审视，我们发现Optional类提供了map、filter和ifPresent方法。这些方法和Streams类中的对应方法有着相似的行为，它们都能以函数式的结构串接计算，

由于库自身提供了缺失值的检测机制，不再需要用户代码的干预。这种进行内部检测还是外部检测的选择和在Stream库中进行内部迭代还是在用户代码中进行外部迭代的选择极其类似。

本节最后我们不再涉及函数式编程的内容，而是要讨论一下Java 8对库的前向兼容性支持，这一技术受到了软件工程发展的推动。

16.1.5　默认方法

Java 8中增加了不少新特性，但是它们一般都不对个体程序的行为带来影响。不过，有一件事情是例外，那就是新增的*默认方法*。接口中新引入的默认方法对类库的设计者而言简直是如鱼得水。Java 8之前，接口主要用于定义方法签名，现在它们还能为接口的使用者提供方法的默认实现，如果接口的设计者认为接口中声明的某个方法并不需要每一个接口的用户显式地提供实现，他就可以考虑在接口的方法声明中为其定义默认方法。

对类库的设计者而言，这是个伟大的新工具，原因很简单，它提供的能力能帮助类库的设计者们定义新的操作，增强接口的能力，类库的用户们（即那些实现该接口的程序员们）不需要花费额外的精力重新实现该方法。因此，默认方法与库的用户也有关系，它们屏蔽了将来的变化对用户的影响。第9章针对这一问题进行了更加深入的探讨。

自此，我们已经完成了对Java 8中新概念的总结。现在我们会转向更为棘手的主题，那就是Java 8之后的版本中可能会有哪些新的改进以及新的特性出现。

16.2　Java 的未来

让我们看看关于Java未来的一些讨论。关于这一主题的大多数内容都会在JDK改进提议（JDK Enhancement Proposal）中进行讨论，它的网址是http://openjdk.java.net/jeps/0。我们在这里想要讨论的主要是一些看起来很合理、实现起来却颇有难度的部分，以及一些由于和现存特性的协作有问题而无法引入到Java中的部分。

16.2.1　集合

Java的发展是一个循序渐进的过程，它从来就不是一蹴而就的。Java中融入了大量伟大的思想，比如：数组取代了集合，之后的Stream又进一步增强了集合的功能。当然，乌龙的情况也偶有发生，有的特性其优势变得更加明显（比如集合之于数组），但我们在做替代时却忽略了被替代特性的一些优点。一个比较典型的例子是容器的初始化。比如，Java中数组可以通过下面这种形式，在声明数组的同时进行初始化：

```
Double [] a = {1.2, 3.4, 5.9};
```

它是以下这种语法的简略形式：

```
Double [] a = new Double[]{1.2, 3.4, 5.9};
```

为处理诸如由数组表示的顺序数据结构，集合（通过`Collection`接口）提供了一种更优秀也更一致的解决方案。不过它们的初始化被忽略了。让我们回想一下你是如何初始化一个`HashMap`的。你只能通过下面这样的代码完成初始化工作：

```
Map<String, Integer> map = new HashMap<>();
map.put("raoul", 23);
map.put("mario", 40);
map.put("alan", 53);
```

你可能更愿意通过下面的方式达到这一目标：

```
Map<String, Integer> map = #{"Raoul" -> 23, "Mario" -> 40, "Alan" -> 53};
```

这里的`#{...}`是一种集合常量，它们代表了集合中的一系列值组成的列表。这似乎是一个毫无争议的特性[①]，不过它当前在Java中还不支持。

16.2.2 类型系统的改进

我们会讨论对Java当前类型系统的两种潜在可能的改进，分别是声明位置变量（declaration-site variance）和本地变量类型推断（local variable type inference）。

1. 声明位置变量

Java加入了对通配符的支持，来更灵活地支持泛型的子类型（subtyping），或者我们可以更通俗地称之为“用户定义变量”（use-site variance）。这也是下面这段代码合法的原因：

```
List<? extends Number> numbers = new ArrayList<Integer>();
```

不过下面的这段赋值（省略了`? extends`）会产生一个编译错误：

```
List<Number> numbers = new ArrayList<Integer>();        <—— 类型不兼容
```

很多编程语言（比如C#和Scala）都支持一种比较独特的变量机制，名为声明位置变量。它们允许程序员们在定义泛型时指定变量。对于天生就为变量的类而言，这一特性尤其有用。比如，`Iterator`就是一个天生的协变量，而`Comparator`则是一个天生的逆变量。使用它们时你无需考虑到底是应该使用`? extends`，还是使用`? super`。这也是说在Java中添加声明位置变量极其有用的原因，因为这些规范会在声明类时就出现。这样一来，程序员的认知负荷就会减少。注意，截至本书写作时（2014年6月），已经有一个提议处于研究过程中，希望能在Java 9中引入声明位置变量[②]。

2. 更多的类型推断

最初在Java中，无论何时我们使用一个变量或方法，都需要同时给出它的类型。例如：

```
double convertUSDToGBP(double money) { ExchangeRate e = ...; }
```

它包含了三种类型；这段代码给出了函数`convertUSDToGBP`的结果类型，它的参数`money`的类

① 当前的Java新特性提议请参考http://openjdk.java.net/jeps/186。

② 参见https://bugs.openjdk.java.net/browse/JDK-8043488。

型，以及方法使用的本地变量e的类型。随着时间的推移，这种限制被逐渐放开了。首先，你可以在一个表达式中忽略泛型参数的类型，通过上下文决定其类型。比如：

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

这段代码在Java 7之后可以缩略为：

```
Map<String, List<String>> myMap = new HashMap<>();
```

其次，利用同样的思想，你可以将由上下文决定的类型交由一个表达式决定，即由Lambda表达式来决定，比如：

```
Function<Integer, Boolean> p = (Integer x) -> booleanExpression;
```

省略类型后，这段代码可以精简为：

```
Function<Integer, Boolean> p = x -> booleanExpression;
```

这两种情况都是由编译器对省略的类型进行推断的。

如果一种类型仅包含单一的标识符，类型推断能带来一系列的好处，其中比较主要的一点是，用一种类型替换另一种可以减少编辑工作量。不过，随着类型数量的增加，出现了由更加泛型的类型参数化的泛型，这时类型推断就带来了新的价值，它能帮助我们改善程序的可读性。[①]

Scala和C#中都允许使用关键词var替换本地变量的初始化声明，编译器会依据右边的变量填充恰当的类型。比如，我们之前展示过的使用Java语法的myMap声明可以像下面这样改写：

```
var myMap = new HashMap<String, List<String>>();
```

这种思想被称为本地变量类型推断，你可能期待Java中也提供类似的特性，因为它能消除冗余的类型，减少杂乱的代码。

然而，它也可能受到一些质疑，比如，类Car继承类Vehicle后，你进行了下面这样的声明：

```
var x = new Vehicle();
```

那么，你到底期望x的类型为Car还是Vehicle呢？这个例子中，一个简单的解释就能解决问题，即缺失的类型就是初始化器对象的类型（这里为Vehicle），由此我们可以得出一个结论，没有初始化器时，不要使用var声明对象。

16.2.3　模式匹配

我们曾经在第14章中讨论过，函数式语言通常都会提供某种形式的模式匹配——作为switch的一种改良形式。通过这种模式匹配，你可以查询“这个值是某个类的实例吗”，或者你也可以选择递归地查询某个字段是否包含了某些值。

① 当然，以一种直观的方式进行类型推断也是非常重要的。类型推断最适合的情况是只存在一种可能性，或者一种比较容易文档化的方式，借此重建用户省略的类型。如果系统推断出的类型与用户最初设想的类型并不一致，就会带来很多问题；所以良好的类型推断设计在面临两种不可比较的类型时，都会给出一个默认的类型，利用默认类型来避免出现随机选择错误的类型。

我们有必要提醒你，即使是传统的面向对象设计也已经不推荐使用switch了，现在大家更推荐的方式是采用一些设计模式，比如访问者模式，使用访问者模式时，程序利用dispatch方法，依据数据类型来选择相应的控制流，不再使用传统的switch方式。这并非另一种编程语言中的事——函数式编程语言中使用基于数据类型的模式匹配通常也是设计程序最便捷的方式。

将类Scala的模式匹配全盘地移植到Java中似乎是个巨大的工程，但是基于switch语法最近的泛化（switch现在已经不再局限于只允许对String进行操作），你可以想象更加现代的语法扩展会有哪些。现在，凭借instanceof，你可以通过switch直接对对象进行操作。这里，我们会对14.4节中的示例进行重构，假设有这样一个类Expr，它有两个子类，分别是BinOp和Number：

```
switch (someExpr) {
      case (op instanceof BinOp):
         doSomething(op.opname, op.left, op.right);
      case (n instanceof Number):
         dealWithLeafNode(n.val);
      default:
         defaultAction(someExpr);
}
```

这里有几点需要特别注意。我们在case (op instanceof BinOp):这段代码中借用了模式匹配的思想，op是一个新的局部变量（类型为BinOp），它和SomeExpr都绑定到了同一个值；类似地，在Number的case判断中，n被转化为了Number类型的变量。而默认情况不需要进行任何变量绑定。和采用串接的if-then-else加子类型转换比起来，这种实现方式避免了大量的模板代码。习惯了传统面向对象方式的设计者很可能会说如果采用访问者模式在子类型中实现这种“数据类型”式的分派，表达的效果会更好，不过从函数式编程的角度看，后者会导致相关代码散落于多个类的定义中，也不太理想。这是一种典型的设计二分法（design dichotomy）问题，经常会在技术粉间挑起以“表达问题”（expression problem）[①]为幌子的口舌之争。

16.2.4 更加丰富的泛型形式

本节会讨论Java泛型的两个局限性，并探讨可能的解决方案。

1. 具化泛型

Java 5中初次引入泛型时，需要它们尽量保持与现存JVM的后向兼容性。为了达到这一目的，ArrayList<String>和ArrayList<Integer>的运行时表示是相同的。这被称作泛型多态（generic polymorphism）的消除模式（erasure model）。这一选择伴随着一定程度的运行时消耗，不过对于程序员而言，这无关痛痒，影响最大的是传给泛型的参数只能为对象类型。如果Java支持ArrayList<int>这种类型的泛型，那么你就可以在堆上分配由简单数据值构成的ArrayList对象，比如42，不过这样一来ArrayList容器就无法了解它所容纳的到底是一个对象类型的值，比如一个String，还是一个简单的int值，比如42。

① 更加完整的解释请参见http://en.wikipedia.org/wiki/Expression_problem。

某种程度上看，这并没有什么危害——如果你可以从ArrayList<int>中得到简单值42，或者从ArrayList<String>中得到String对象abc，为什么还要担忧ArrayList容器无法辨识呢？非常不幸，答案是垃圾收集，因为一旦缺失了ArrayList中内容的运行时信息，JVM就无法判断ArrayList中的元素13到底是一个Integer的引用（可以被垃圾收集器标记为“in use”并进行跟踪），还是int类型的简单数据（几乎可以说是无法跟踪的）。

C#语言中，ArrayList<String>、ArrayList<Integer>以及ArrayList<int>的运行时表示在原则上就是不同的。即使它们的值是相同的，也伴随着足够的运行时类型信息，这些信息可以帮助垃圾收集器判断一个字段值到底是引用，还是简单数据。这被称为*泛型多态的具化模式*，或*具化泛型*。“具化”这个词意味着“将某些默认隐式的东西变为显式的”。

很明显，具化泛型是众望所归的，它们能将简单数据类型及其对应的对象类型更好地融合——下一节中，你会看到这之前的一些问题。实现具化泛型的主要难点在于，Java需要保持后向兼容性，并且这种兼容需要同时覆盖JVM，以及使用了反射且希望进行泛型清除的遗留代码。

2. 泛型中特别为函数类型增加的语法灵活性

自从被Java 5引入，泛型就证明了其独特的价值。它们还特别适用于表示Java 8中的Lambda类型以及各种方法引用。通过下面这种方式你可以表示使用单一参数的函数：

```
Function<Integer, Integer> square = x -> x * x;
```

如果你有一个使用两个参数的函数，可以采用类型BiFunction<T, U, R>，这里的T表示第一个参数的类型，U表示第二个参数的类型，而R是计算的结果。不过，Java 8中并未提供TriFunction这样的函数，除非你自己声明了一个！

同理，你不能用Function<T, R>引用表示某个不接受任何参数，返回值为R类型的函数；只能通过Supplier<R>达到这一目的。

从本质上来说，Java 8的Lambda极大地拓展了我们的编程能力，但可惜的是，它的类型系统并未跟上代码灵活度提升的脚步。在很多的函数式编程语言中，你可以用(Integer, Double) => String这样的类型实现Java 8中BiFunction<Integer, Double, String>调用得到同样的效果；类似地，可以用Integer => String表示Function<Integer, String>，甚至可以用() => String表示Supplier<String>。你可以将=>符号看作Function、BiFunction、Supplier，以及其他相似函数的中缀表达式版本。我们只需要对现有Java语言的类型格式稍作扩展就能提供Scala语言那样更具可读性的类型，关于Java和Scala的比较我们已经在第15章中详细讨论过了。

3. 原型特化和泛型

在Java语言中，所有的简单数据类型，比如int，都有对应的对象类型（以刚才的例子而言，它是java.lang.Integer）；通常我们把它们称为不装箱类型和装箱类型。虽然这种区分有助于提升运行时的效率，但是这种方式定义的类型也可能带来一些困扰。比如，有人可能会问为什么Java 8中我们需要编写Predicate<Apple>，而不是直接采用Function<Apple, Boolean>

的方式？事实上，`Predicate<Apple>`类型的对象在执行`test`方法调用时，其返回值依旧是简单类型`boolean`。

与此相反，和所有泛型一样，`Function`只能使用对象类型的参数。以`Function<Apple, Boolean>`为例，它使用的是对象类型`Boolean`，而不是简单数据类型`boolean`。所以使用`Predicate<Apple>`更加高效，因为它无需将`boolean`装箱为`Boolean`。因为存在这样的问题，导致类库的设计者在Java时创建了多个类似的接口，比如`LongToIntFunction`和`BooleanSupplier`，而这又进一步增加了大家理解的负担。另一个例子和`void`之间的区别有关，`void`只能修饰不带任何值的方法，而`Void`对象实际包含了一个值，它有且仅有一个`null`值——这是一个经常在论坛上讨论的问题。对于`Function`的特殊情况，比如`Supplier<T>`，你可以用前面建议的新操作符将其改写为`() => T`，这进一步佐证了由于简单数据类型（primitive type）与对象类型（object type）的差异所导致的分歧。我们在之前的内容中已经介绍了怎样通过具化泛型解决这其中的很多问题。

16.2.5 对不变性的更深层支持

Java 8只支持三种类型的值，分别为：

- 简单类型值
- 指向对象的引用
- 指向函数的引用

听我们说起这些，有些专业的读者可能会感到失望。我们在某种程度上会坚持自己的观点，介绍说"现在方法可以使用这些值作为参数，并返回相应的结果了"。不过，我们也承认这其中的确还存在着一定的问题，比如，当你返回一个指向可变数组的引用时，它多大程度上应该是一个（算术）值？很明显，字符串或者不可变数组都是值，不过对于可变对象或者数组而言，情况远非那么泾渭分明——你的方法可能返回一个元素以升序排列的数组，不过另一些代码可能在之后对其中的某些元素进行修改。

如果我们想在Java中真正实现函数式编程，那么语言层面的支持就必不可少了，比如"不可变值"。正如我们在第13章中所了解的那样，关键字`final`并未在真正意义上是要达到这一目标，它仅仅避免了对它所修饰字段的更新。我们看看下面这个例子：

```
final int[] arr = {1, 2, 3};
final List<T> list = new ArrayList<>();
```

前者禁止了直接的赋值操作`arr = ...`，不过它并未阻止以`arr[1]=2`这样的方式对数组进行修改；而后者禁止了对列表的赋值操作，但并未禁止以其他方法修改列表中的元素！关键字`final`对于简单数据类型的值操作效果很好，不过对于对象引用，它通常只是一种错误的安全感。

那么我们该如何解决这一问题呢？由于函数式编程对不能修改现存数据结构有非常严格的要求，所以它提供了一个更强大的关键字，比如`transitively_final`，该关键字用于修饰引用类型的字段，确保无论是直接对该字段本身的修改，还是对通过该字段能直接或间接访问到的

对象的修改都不会发生。

这些类型体现了关于值的一个理念：变量值是不可修改的，只有变量（它们存储着具体的值）可以被修改，修改之后变量中包含了其他一些不可变值。正如我们在本节开头所提及的，Java的作者，包括我们，时不时地都喜欢针对Java中值与可变数组的转化展开讨论。接下来的一节，我们会讨论一下值类型（value type），声明为值类型的变量只能包含不可变值，然而，除非使用了`final`关键词进行修饰，否则变量的值还是能够进行更新。

16.2.6 值类型

这一节，我们会讨论简单数据类型和对象类型之间的差异，并结合前文针对值类型的讨论，希望能借此帮助你以函数式的方式进行编程，就像对象类型是面向对象编程不可缺失的一环那样。我们讨论的很多问题都是相互交织的，所以，很难以区隔的方式解释某一个单独的问题。所以，我们会从不同的角度定位这些问题。

1. 为什么编译器不能对`Integer`和`int`一视同仁

自从Java 1.1版本以来，Java语言逐渐具备了隐式地进行装箱和拆箱的能力，你可能会问现在是否是一个恰当的时机，让Java语言一视同仁地处理简单数据类型和对象数据类型，比如将`Integer`和`int`同等对待，依赖Java编译器将它们优化为JVM最适合的形式。

这个想法在原则上是非常美好的，不过让我们看看在Java中添加`Complex`类型后会引发哪些问题，以及为什么装箱会导致这样的问题。用于建模复数的`Complex`包含了两个部分，分别是实数（real）和虚数（imaginary），一种很直观的定义如下：

```
class Complex {
    public final double re;
    public final double im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
     public static Complex add(Complex a, Complex b) {
        return new Complex(a.re+b.re, a.im+b.im);
     }
}
```

不过类型`Complex`的值为引用类型，对`Complex`的每个操作都需要进行对象分配——增加了`add`中两次加法操作的开销。我们需要的是类似`Complex`的简单数据类型，我们也许可以称其为`complex`。

这里的问题是我们想要一种“不装箱的对象”，可是无论Java还是JVM，对此都没有实质的支持。至此，我们只能悲叹了，“噢，当然编译器可以对它进行优化”。坏消息是，这远比看起来要复杂得多；虽然Java带有基于名为“逃逸分析”的编译器优化（这一技术自Java 1.1版本开始就已经有了），它能在某些时候判断拆箱的结果是否正确，然而其能力依旧有一定的限制，它受制于Java对对象类型的判断。以下面的这个难题为例：

```
double d1 = 3.14;
double d2 = d1;
Double o1 = d1;
Double o2 = d2;
Double ox = o1;
System.out.println(d1 == d2 ? "yes" : "no");
System.out.println(o1 == o2 ? "yes" : "no");
System.out.println(o1 == ox ? "yes" : "no");
```

最后这段代码输出的结果为“yes”“no”“yes”。专业的Java程序员可能会说“多愚蠢的代码，每个人都知道最后这两行你应该使用equals而不是==”。不过，请允许我们继续用这个例子进行说明。虽然所有这些简单变量和对象都保存了不可变值3.14，实际上也应该是没有差别的，但是由于有对o1和o2的定义，程序会创建新的对象，而==操作符（利用特征比较）可以将这二者区分开来。请注意，对于简单变量，特征比较采用的是逐位比较（bitwise comparison），对于对象类型它采用的是引用比较（reference equality）。因此，很多时候由于编译器需要遵守对象的语义，我们随机创建的新的Double对象（Double对象继承自Object）也需要遵守该语义。你之前见过这些讨论，无论是较早的时候关于值对象的讨论，还是第14章围绕更新持久化数据结构保证引用透明性的方法讨论。

2. 值对象——无论简单类型还是对象类型都不能包打天下

关于这个问题，我们建议的解决方案是重新回顾一下Java的初心：(1) 任何事物，如果不是简单数据类型，就是对象类型，所有的对象类型都继承自Object；(2) 所有的引用都是指向对象的引用。

事情的发展是这样开始的。Java中有两种类型的值：一类是对象类型，它们包含着可变的字段（除非使用了final关键字进行修饰），对这种类型值的特征，可以使用==进行比较；还有一类是值类型，这种类型的变量是不能改变的，也不带任何的引用特征（reference identity），简单类型就属于这种更宽泛意义上的值类型。这样，我们就能创建用户自定义值的类型了（这种类型的变量推荐小写字符开头，突出它们与int和boolean这类简单类型的相似性）。对于值类型，默认情况下，硬件对int进行比较时会以一个字节接着一个字节逐次的方式进行，==会以同样的方式一个元素接着一个元素地对两个变量进行比较。你可以将这看成对浮点比较的覆盖，不过这里会进行一些更加复杂的操作。Complex是一个绝佳的例子用于介绍非简单类型的值；它们和C#中的结构struct极其类似。

此外，值类型可以减少对存储的要求，因为它们并不包含引用特征。图16-1引用了容量为3的一个数组，其中的元素0、1和2分别用淡灰、白色和深灰色标记。左边的图展示了一种比较典型的存储需求，其中的Pair和Complex都是对象类型，而右边展示的是一种更优的布局，这里的Pair和Complex都是值类型（注意，我们在这里特意使用了小写的pair和complex，目的就是想强调它们与简单类型的相似性）。也请注意，值类型极有可能提供更好的性能，无论是数据访问（用单一的索引地址指令替换多层的指针转换），还是对硬件缓存的利用率（因为数据现在采用的是连续存储）。

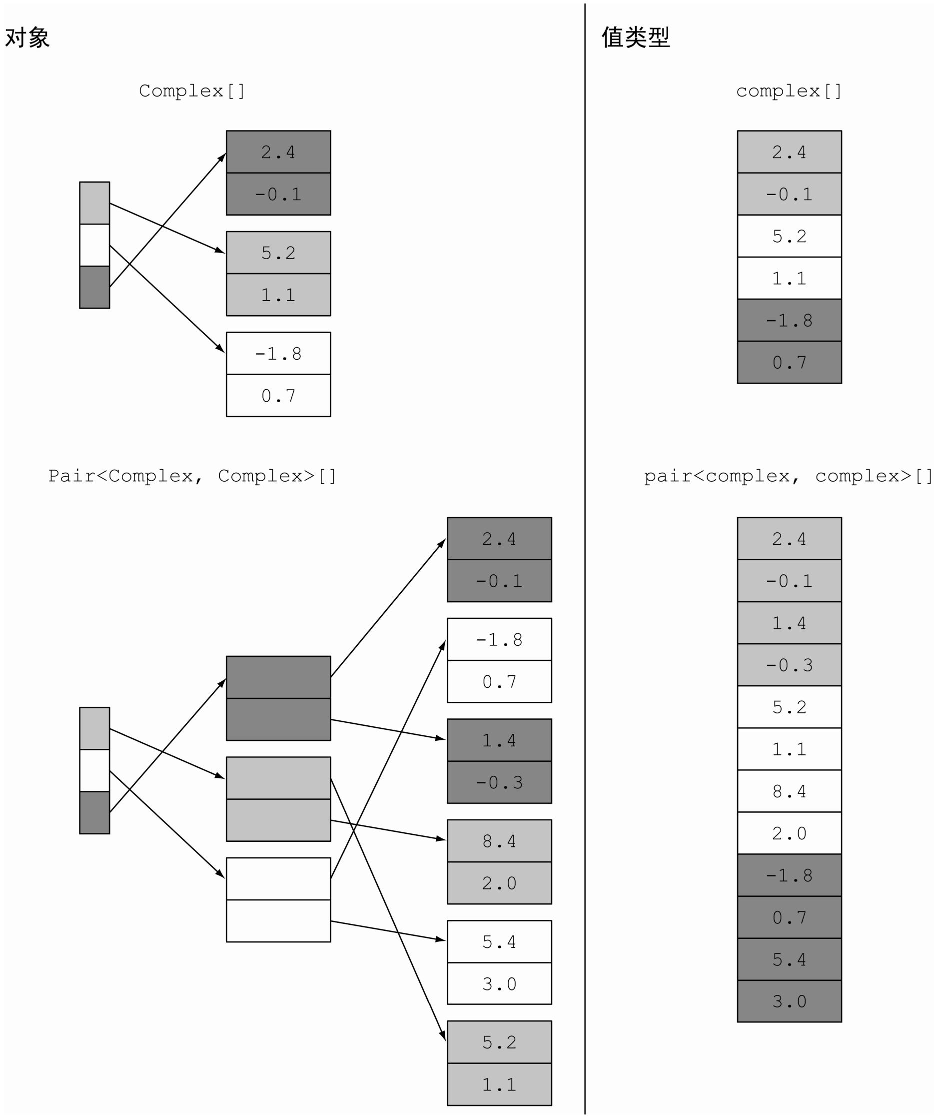


图16-1　对象与值类型

注意，由于值类型并不包含引用特征，编译器可以随意对它们进行装箱和拆箱。如果你将一个complex变量作为参数从一个函数传递给另一个函数，编译器可以很自然地将它们拆分为两个单独的double类型的参数。（由于JVM只提供了以64位寄存器传递值的方法返回指令，所以在JVM中要实现不装箱，直接返回是比较复杂的。）不过，如果你传递一个很大的值作为参数（比如说一个很大的不可变数组），那么编译器可以以透明的方式（透明于用户），对其进行装箱，将

其转化为一个引用进程传递。类似的技术已经在C#中存在；下面引用了一段微软的介绍①：

> 结构看起来和类十分相似，但是二者之间存在重大差异，你应该了解它们之间的不同。首先，类[这里指的是C#中的类]属于引用类型，而结构（struct）属于值类型。使用结构，你可以创建对象[比如sic]，它的行为就像那些内置[简单]类型一样，享受同等的待遇。

截至本书写作时（2014年6月），Java也已经接受了一份采用值类型的具体建议②。

3. 装箱、泛型、值类型——互相交织的问题

我们希望能够在Java中引入值类型，因为函数式编程处理的不可变对象并不含有特征。我们希望简单数据类型可以作为值类型的特例，但又不要有当前Java所携带的泛型的消除模式，因为这意味着值类型不做装箱就不能使用泛型。由于对象的消除模式，简单类型（比如`int`）的对象（装箱）版本（比如`Integer`）对集合和Java泛型依旧非常重要，不过它们继承自`Object`（并因此引用相等），这被当成了一种缺点。解决这些问题中的任何一个就意味着解决了所有的问题。

16.3　写在最后的话

本书探索了Java 8新增加的一系列新特性；它们所代表的可能是自Java创建以来最大的一次演进——唯一可以与之相提并论的大的演进也是在10年之前，即Java 5中所引入的泛型。这一章里我们还了解了Java进一步发展所面临的压力。用一句话来总结，我们会说：

> Java 8已经占据了一个非常好的位置，可以暂时歇口气，但这绝不是终点！

我们希望你能享受这一段Java 8的探索旅程，也希望本书能燃起你对了解函数式编程及Java 8进一步发展的兴趣。

---

① 如需了解结构语法和使用，以及类与结构之间的差异，请访问http://msdn.microsoft.com/en-us/library/aa288471(v=vs.71).aspx。

② John Rose等，“值的状态”，2014年4月初始版本，http://cr.openjdk.java.net/~jrose/values/values-0.html。

附录 A

其他语言特性的更新

本附录中，我们会讨论Java 8中其他的三个语言特性的更新，分别是重复注解（repeated annotation）、类型注解（type annotation）和通用目标类型推断（generalized target-type inference）。附录B会讨论Java 8中类库的更新。我们不会涉及JDK 8中的所有内容，比如我们不会谈及Nashorn或者是精简运行时（Compact Profiles），因为它们属于JVM的新特性。本书专注于介绍类库和语言的更新。如果你对Nashorm或者精简运行时感兴趣，我们推荐你阅读以下两个链接的内容，分别是http://openjdk.java.net/projects/nashorn/和http://openjdk.java.net/jeps/161。

A.1 注解

Java 8在两个方面对注解机制进行了改进，分别为：

- 你现在可以定义重复注解
- 使用新版Java，你可以为任何类型添加注解

正式开始介绍之前，我们先快速地回顾一下Java 8之前的版本能用注解做什么，这有助于我们加深对新特性的理解。

Java中的*注解*是一种对程序元素进行配置，提供附加信息的机制（注意，在Java 8之前，只有声明可以被注解）。换句话说，它是某种形式的*语法元数据*（syntactic metadata）。比如，注解在JUnit框架中就使用得非常频繁。下面这段代码中，`setUp`方法使用了`@Before`进行注解，而`testAlgorithm`使用了`@Test`进行注解：

```
@Before
public void setUp(){
    this.list = new ArrayList<>();
}

@Test
public void testAlgorithm(){
     ...
    assertEquals(5, list.size());
}
```

注解尤其适用于下面这些场景。

- 在JUnit的上下文中，使用注解能帮助区分哪些方法用于单元测试，哪些用于做环境搭建工作。
- 注解可以用于文档编制。比如，`@Deprecated`注解被广泛应用于说明某个方法不再推荐使用。
- Java编译器还可以依据注解检查错误，禁止报警输出，甚至还能生成代码。
- 注解在Java企业版中尤其流行，它们经常用于配置企业应用程序。

A.1.1 重复注解

之前版本的Java禁止对同样的注解类型声明多次。由于这个原因，下面的第二句代码是无效的：

```
@interface Author { String name(); }

@Author(name="Raoul") @Author(name="Mario") @Author(name="Alan")
class Book{ }
```

错误：重复的注解

Java企业版的程序员经常通过一些惯用法绕过这一限制。你可以声明一个新的注解，它包含了你希望重复的注解数组。这种方法的形式如下：

```
@interface Author { String name(); }
@interface Authors {
    Author[] value();
}

@Authors(
  { @Author(name="Raoul"), @Author(name="Mario") , @Author(name="Alan")}
)
class Book{}
```

`Book`类的嵌套注解相当难看。这就是Java 8想要从根本上移除这一限制的原因，去掉这一限制后，代码的可读性会好很多。现在，如果你的配置允许重复注解，你可以毫无顾虑地一次声明多个同一种类型的注解。它目前还不是默认行为，你需要显式地要求进行重复注解。

创建一个重复注解

如果一个注解在设计之初就是可重复的，你可以直接使用它。但是，如果你提供的注解是为用户提供的，那么就需要做一些工作，说明该注解可以重复。下面是你需要执行的两个步骤：

(1) 将注解标记为`@Repeatable`

(2) 提供一个注解的容器

下面的例子展示了如何将`@Author`注解修改为可重复注解：

```
@Repeatable(Authors.class)
@interface Author { String name(); }
@interface Authors {
    Author[] value();
}
```

完成了这样的定义之后，Book类可以通过多个@Author注解进行注释，如下所示：

```
@Author(name="Raoul") @Author(name="Mario") @Author(name="Alan")
class Book{ }
```

编译时，Book会被认为使用了@Authors({@Author(name="Raoul"), @Author(name=”Mario”), @Author(name=”Alan”)})这样的形式进行了注解，所以，你可以把这种新的机制看成是一种语法糖，它提供了Java程序员之前利用的惯用法类似的功能。为了确保与反射方法在行为上的一致性，注解会被封装到一个容器中。Java API中的getAnnotation(Class<T> annotation-Class)方法会为注解元素返回类型为T的注解。如果实际情况有多个类型为T的注解，该方法的返回到底是哪一个呢？

我们不希望一下子就陷入细节的魔咒，类Class提供了一个新的getAnnotationsByType方法，它可以帮助我们更好地使用重复注解。比如，你可以像下面这样打印输出Book类的所有Author注解：

```
public static void main(String[] args) {
    Author[] authors = Book.class.getAnnotationsByType(Author.class);   ← 返回一个由重复注解Author组成的数组
    Arrays.asList(authors).forEach(a -> { System.out.println(a.name()); });
}
```

这段代码要正常工作的话，需要确保重复注解及它的容器都有运行时保持策略。关于与遗留反射方法的兼容性的更多讨论，可以参考http://cr.openjdk.java.net/~abuckley/8misc.pdf。

A.1.2　类型注解

从Java 8开始，注解已经能应用于任何类型。这其中包括new操作符、类型转换、instanceof检查、泛型类型参数，以及implements和throws子句。这里，我们举了一个例子，这个例子中类型为String的变量name不能为空，所以我们使用了@NonNull对其进行注解：

```
@NonNull String name = person.getName();
```

类似地，你可以对列表中的元素类型进行注解：

```
List<@NonNull Car> cars = new ArrayList<>();
```

为什么这么有趣呢？实际上，利用好对类型的注解非常有利于我们对程序进行分析。这两个例子中，通过这一工具我们可以确保getName不返回空，cars列表中的元素总是非空值。这会极大地帮助你减少代码中不期而至的错误。

Java 8并未提供官方的注解或者一种工具能以开箱即用的方式使用它们。它仅仅提供了一种功能，你使用它可以对不同的类型添加注解。幸运的是，这个世界上还存在一个名为Checker的框架，它定义了多种类型注解，使用它们你可以增强类型检查。如果对此感兴趣，我们建议你看看它的教程，地址链接为：http://www.checker-framework.org。关于在代码中的何处使用注解的更多内容，可以访问http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7.4。

A.2　通用目标类型推断

Java 8对泛型参数的推断进行了增强。相信你对Java 8之前版本中的类型推断已经比较熟悉了。比如，Java中的方法emptyList方法定义如下：

```
static <T> List<T> emptyList();
```

emptyList方法使用了类型参数T进行参数化。你可以像下面这样为该类型参数提供一个显式的类型进行函数调用：

```
List<Car> cars = Collections.<Car>emptyList();
```

不过Java也可以推断泛型参数的类型。上面的代码和下面这段代码是等价的：

```
List<Car> cars = Collections.emptyList();
```

Java 8出现之前，这种推断机制依赖于程序的上下文（即目标类型），具有一定的局限性。比如，下面这种情况就不大可能完成推断：

```
static void cleanCars(List<Car> cars) {
}
cleanCars(Collections.emptyList());
```

你会遭遇下面的错误：

```
cleanCars (java.util.List<Car>)cannot be applied to
     (java.util.List<java.lang.Object>)
```

为了修复这一问题，你只能像我们之前展示的那样提供一个显式的类型参数。

Java 8中，目标类型包括向方法传递的参数，因此你不再需要提供显式的泛型参数：

```
List<Car> cleanCars = dirtyCars.stream()
                               .filter(Car::isClean)
                               .collect(Collectors.toList());
```

通过这段代码，我们能很清晰地了解到，正是伴随Java 8而来的改进让你只需要一句Collectors.toList()就能完成期望的工作，不再需要编写像Collectors.<Car>toList()这么复杂的代码了。

附录 B 类库的更新

本附录会审视Java 8方法库中重要的更新。

B.1 集合

Collection API在Java 8中最重大的更新就是引入了流，我们已经在第4章到6章进行了介绍。当然，除此之外，Collection API还有一部分更新，本附录会简要地讨论。

B.1.1 其他新增的方法

Java API的设计者们充分利用默认方法，为集合接口和类新增了多个新的方法。这些新增的方法我们已经列在表B-1中了。

表B-1 集合类和接口中新增的方法

类/接口	新 方 法
`Map`	`getOrDefault`，`forEach`，`compute`，`computeIfAbsent`，`computeIfPresent`，`merge`，`putIfAbsent`，`remove(key,value)`，`replace`，`replaceAll`
`Iterable`	`forEach`，`spliterator`
`Iterator`	`forEachRemaining`
`Collection`	`removeIf`，`stream`，`parallelStream`
`List`	`replaceAll`，`sort`
`BitSet`	`stream`

1. **`Map`**

`Map`接口的变化最大，它增加了多个新方法，利用这些新方法能更加便利地操纵`Map`中的数据。比如，`getOrDefault`方法就可以替换现在检测`Map`中是否包含给定键映射的惯用方法。如果`Map`中不存在这样的键映射，你可以提供一个默认值，方法会返回该默认值。使用之前版本的Java，要实现这一目的，你可能会如下编这段代码：

```
Map<String, Integer> carInventory = new HashMap<>();
Integer count = 0;
if(map.containsKey("Aston Martin")){
```

```
    count = map.get("Aston Martin");
}
```

使用新的Map接口之后，你只需要简单地编写一行代码就能实现这一功能，代码如下：

```
Integer count = map.getOrDefault("Aston Martin", 0);
```

注意，这一方法仅在没有映射时才生效。比如，如果键被显式地映射到了空值，那么该方法是不会返回你设定的默认值的。

另一个特别有用的方法是computeIfAbsent，这个方法在第14章解释记忆表时曾经简要地提到过。它能帮助你非常方便地使用缓存模式。比如，我们假设你需要从不同的网站抓取和处理数据。这种场景下，如果能够缓存数据是非常有帮助的，这样你就不需要每次都执行（代价极高的）数据抓取操作了：

```
public String getData(String url){
    String data = cache.get(url);          <-- 检查数据是否已经缓存
    if(data == null){
        data = getData(url);
        cache.put(url, data);          <-- 如果数据没有缓存，那就访问网站抓取数据，紧接着对Map中的数据进行缓存，以备将来使用之需
    }
    return data;
}
```

这段代码，你现在可以通过computeIfAbsent用更加精炼的方式实现，代码如下所示：

```
public String getData(String url){
    return cache.computeIfAbsent(url, this::getData);
}
```

上面介绍的这些方法，其更详细的内容都能在Java API的官方文档中找到[①]。注意，ConcurrentHashMap也进行了更新，提供了新的方法。我们会在B.2节讨论。

2. 集合

removeIf方法可以移除集合中满足某个谓词的所有元素。注意，这一方法与我们在介绍Stream API时提到的filter方法不大一样。Stream API中的filter方法会产生一个新的流，不会对当前作为数据源的流做任何变更。

3. 列表

replaceAll方法会对列表中的每一个元素执行特定的操作，并用处理的结果替换该元素。它的功能和Stream中的map方法非常相似，不过replaceAll会修改列表中的元素。与此相反，map方法会生成新的元素。

比如，下面这段代码会打印输出[2,4,6,8,10]，因为列表中的元素被原地修改了：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.replaceAll(x -> x * 2);
System.out.println(numbers);          <-- 打印输出[2, 4, 6, 8, 10]
```

① 更多细节请参考http://docs.oracle.com/javase/8/docs/api/java/util/Map.html。

B.1.2 Collections 类

Collections类已经存在了很长的时间，它的主要功能是操作或者返回集合。Java 8中它又新增了一个方法，该方法可以返回不可修改的、同步的、受检查的或者是空的NavigableMap或NavigableSet。除此之外，它还引入了checkedQueue方法，该方法返回一个队列视图，可以扩展进行动态类型检查。

B.1.3 Comparator

Comparator接口现在同时包含了默认方法和静态方法。你可以使用第3章中介绍的静态方法Comparator.comparing返回一个Comparator对象，该对象提供了一个函数可以提取排序关键字。

新的实例方法包含了下面这些。

- reversed——对当前的Comparator对象进行逆序排序，并返回排序之后新的Comparator对象。
- thenComparing——当两个对象相同时，返回使用另一个Comparator进行比较的Comparator对象。
- thenComparingInt、thenComparingDouble、thenComparingLong——这些方法的工作方式和thenComparing方法类似，不过它们的处理函数是特别针对某些基本数据类型（分别对应于ToIntFunction、ToDoubleFunction和ToLongFunction）的。

新的静态方法包括下面这些。

- comparingInt、comparingDouble、comparingLong——它们的工作方式和comparing类似，但接受的函数特别针对某些基本数据类型（分别对应于ToIntFunction、ToDoubleFunction和ToLongFunction）。
- naturalOrder——对Comparable对象进行自然排序，返回一个Comparator对象。
- nullsFirst、nullsLast——对空对象和非空对象进行比较，你可以指定空对象（null）比非空对象（non-null）小或者比非空对象大，返回值是一个Comparator对象。
- reverseOrder——和naturalOrder().reversed()方法类似。

B.2 并发

Java 8中引入了多个与并发相关的更新。首当其冲的当然是并行流，我们在第7章详细讨论过。另外一个就是第11章中介绍的CompletableFuture类。

除此之外，还有一些值得注意的更新。比如，Arrays类现在支持并发操作了。我们会在B.3节讨论这些内容。

这一节，我们想要围绕java.util.concurrent.atomic包的更新展开讨论。这个包的主要功能是处理原子变量（atomic variable）。除此之外，我们还会讨论ConcurrentHashMap类的更新，它现在又新增了几个方法。

B.2.1　原子操作

java.util.concurrent.atomic包提供了多个对数字类型进行操作的类，比如Atomic-Integer和AtomicLong，它们支持对单一变量的原子操作。这些类在Java 8中新增了更多的方法支持。

- getAndUpdate——以原子方式用给定的方法更新当前值，并返回变更之前的值。
- updateAndGet——以原子方式用给定的方法更新当前值，并返回变更之后的值。
- getAndAccumulate——以原子方式用给定的方法对当前及给定的值进行更新，并返回变更之前的值。
- accumulateAndGet——以原子方式用给定的方法对当前及给定的值进行更新，并返回变更之后的值。

下面的例子向我们展示了如何以原子方式比较一个现存的原子整型值和一个给定的观测值（比如10），并将变量设定为二者中较小的一个。

```
int min = atomicInteger.accumulateAndGet(10, Integer::min);
```

Adder和Accumulator

多线程的环境中，如果多个线程需要频繁地进行更新操作，且很少有读取的动作（比如，在统计计算的上下文中），Java API文档中推荐大家使用新的类LongAdder、LongAccumulator、Double-Adder以及DoubleAccumulator，尽量避免使用它们对应的原子类型。这些新的类在设计之初就考虑了动态增长的需求，可以有效地减少线程间的竞争。

LongAddr和DoubleAdder类都支持加法操作，而LongAccumulator和DoubleAccu-mulator可以使用给定的方法整合多个值。比如，可以像下面这样使用LongAdder计算多个值的总和。

代码清单B-1　使用LongAdder计算多个值之和

```
LongAdder adder = new LongAdder();    ← 使用默认构造器，初始的sum值被置为0
adder.add(10);                        ← 在多个不同的线程中进行加法运算
// …
long sum = adder.sum();               ← 到某个时刻得出sum的值
```

或者，你也可以像下面这样使用LongAccumulator实现同样的功能。

代码清单B-2　使用LongAccumulator计算多个值之和

```
LongAccumulator acc = new LongAccumulator(Long::sum, 0);
acc.accumulate(10);                   ← 在几个不同的线程中累计计算值
// …
long result = acc.get();              ← 在某个时刻得出结果
```

B.2.2　ConcurrentHashMap

ConcurrentHashMap类的引入极大地提升了HashMap现代化的程度，新引入的Concurrent-

HashMap对并发的支持非常友好。ConcurrentHashMap允许并发地进行新增和更新操作，因为它仅对内部数据结构的某些部分上锁。因此，和另一种选择，即同步式的Hashtable比较起来，它具有更高的读写性能。

1. 性能

为了改善性能，要对ConcurrentHashMap的内部数据结构进行调整。典型情况下，map的条目会被存储在桶中，依据键生成哈希值进行访问。但是，如果大量键返回相同的哈希值，由于桶是由List实现的，它的查询复杂度为$O(n)$，这种情况下性能会恶化。在Java 8中，当桶过于臃肿时，它们会被动态地替换为排序树（sorted tree），新的数据结构具有更好的查询性能（排序树的查询复杂度为$O(\log(n))$）。注意，这种优化只有当键是可以比较的（比如String或者Number类）时才可能发生。

2. 类流操作

ConcurrentHashMap支持三种新的操作，这些操作和你之前在流中所见的很像：

- forEach——对每个键值对进行特定的操作
- reduce——使用给定的精简函数（reduction function），将所有的键值对整合出一个结果
- search——对每一个键值对执行一个函数，直到函数的返回值为一个非空值

以上每一种操作都支持四种形式，接受使用键、值、Map.Entry以及键值对的函数：

- 使用键和值的操作（forEach、reduce、search）
- 使用键的操作（forEachKey、reduceKeys、searchKeys）
- 使用值的操作 （forEachValue、reduceValues、searchValues）
- 使用Map.Entry对象的操作（forEachEntry、reduceEntries、searchEntries）

注意，这些操作不会对ConcurrentHashMap的状态上锁。它们只会在运行过程中对元素进行操作。应用到这些操作上的函数不应该对任何的顺序，或者其他对象，抑或在计算过程发生变化的值，有依赖。

除此之外，你需要为这些操作指定一个并发阈值。如果经过预估当前map的大小小于设定的阈值，操作会顺序执行。使用值1开启基于通用线程池的最大并行。使用值Long.MAX_VALUE设定程序以单线程执行操作。

下面这个例子中，我们使用reduceValues试图找出map中的最大值：

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
Optional<Integer> maxValue =
    Optional.of(map.reduceValues(1, Integer::max));
```

注意，对int、long和double，它们的reduce操作各有不同（比如reduceValuesToInt、reduceKeysToLong等）。

3. 计数

ConcurrentHashMap类提供了一个新的方法，名叫mappingCount，它以长整型long返回map中映射的数目。我们应该尽量使用这个新方法，而不是老的size方法，size方法返回的类型为int。这是因为映射的数量可能是int无法表示的。

4. 集合视图

ConcurrentHashMap类还提供了一个名为KeySet的新方法，该方法以Set的形式返回ConcurrentHashMap的一个视图（对map的修改会反映在该Set中，反之亦然）。你也可以使用新的静态方法newKeySet，由ConcurrentHashMap创建一个Set。

B.3　Arrays

Arrays类提供了不同的静态方法对数组进行操作。现在，它又包括了四个新的方法（它们都有特别重载的变量）。

B.3.1　使用 parallelSort

parallelSort方法会以并发的方式对指定的数组进行排序，你可以使用自然顺序，也可以为数组对象定义特别的Comparator。

B.3.2　使用 setAll 和 parallelSetAll

setAll和parallelSetAll方法可以以顺序的方式也可以用并发的方式，使用提供的函数计算每一个元素的值，对指定数组中的所有元素进行设置。该函数接受元素的索引，返回该索引元素对应的值。由于parallelSetAll需要并发执行，所以提供的函数必须没有任何副作用，就如第7章和第13章中介绍的那样。

举例来说，你可以使用setAll方法生成一个值为0, 2, 4, 6, …的数组：

```
int[] evenNumbers = new int[10];
Arrays.setAll(evenNumbers, i -> i * 2);
```

B.3.3　使用 parallelPrefix

parallelPrefix方法以并发的方式，用用户提供的二进制操作符对给定数组中的每个元素进行累积计算。通过下面这段代码，你会得到这样的一些值：1, 2, 3, 4, 5, 6, 7, …。

代码清单B-3　使用parallelPrefix并发地累积数组中的元素

```
int[] ones = new int[10];
Arrays.fill(ones, 1);
Arrays.parallelPrefix(ones, (a, b) -> a + b);  <-- ones现在的内容是[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

B.4　Number 和 Math

Java 8 API对Number和Math也做了改进，为它们增加了新的方法。

B.4.1 Number

Number类中新增的方法如下。

- Short、Integer、Long、Float和Double类提供了静态方法sum、min和max。在第5章介绍reduce操作时，你已经见过这些方法。
- Integer和Long类提供了compareUnsigned、divideUnsigned、remainderUnsigned和toUnsignedLong方法来处理无符号数。
- Integer和Long类也分别提供了静态方法parseUnsignedInt和parseUnsignedLong将字符解析为无符号int或者long类型。
- Byte和Short类提供了toUnsignedInt和toUnsignedLong方法通过无符号转换将参数转化为int或者long类型。类似地，Integer类现在也提供了静态方法toUnsignedLong。
- Double和Float类提供了静态方法isFinite，可以检查参数是否为有限浮点数。
- Boolean类现在提供了静态方法logicalAnd、logicalOr和logicalXor，可以在两个boolean之间执行and、or和xor操作。
- BigInteger类提供了byteValueExact、shortValueExact、intValueExact和longValueExact，可以将BigInteger类型的值转换为对应的基础类型。不过，如果在转换过程中有信息的丢失，方法会抛出算术异常。

B.4.2 Math

如果Math中的方法在操作中出现溢出，Math类提供了新的方法可以抛出算术异常。支持这一异常的方法包括使用int和long参数的addExact、subtractExact、multipleExact、incrementExact、decrementExact和negateExact。此外，Math类还新增了一个静态方法toIntExact，可以将long值转换为int值。其他的新增内容包括静态方法floorMod、floorDiv和nextDown。

B.5 Files

Files类最引人注目的改变是，你现在可以用文件直接产生流。第5章中提到过新的静态方法Files.lines，通过该方法你可以以延迟方式读取文件的内容，并将其作为一个流。此外，还有一些非常有用的静态方法可以返回流。

- Files.list——生成由指定目录中所有条目构成的Stream<Path>。这个列表不是递归包含的。由于流是延迟消费的，处理包含内容非常庞大的目录时，这个方法非常有用。
- Files.walk——和Files.list有些类似，它也生成包含给定目录中所有条目的Stream<Path>。不过这个列表是递归的，你可以设定递归的深度。注意，该遍历是依照深度优先进行的。

- ❑ Files.find——通过递归地遍历一个目录找到符合条件的条目，并生成一个Stream<Path>对象。

B.6 Reflection

附录A中已经讨论过Java 8中注解机制的几个变化。Reflection API的变化就是为了支撑这些改变。

除此之外，Relection接口的另一个变化是新增了可以查询方法参数信息的API，比如，你现在可以使用新增的java.lang.reflect.Parameter类查询方法参数的名称和修饰符，这个类被新的java.lang.reflect.Executable类所引用，而java.lang.reflect.Executable通用函数和构造函数共享的父类。

B.7 `String`

String类也新增了一个静态方法，名叫join。你大概已经猜出它的功能了，它可以用一个分隔符将多个字符串连接起来。你可以像下面这样使用它：

```
String authors = String.join(", ", "Raoul", "Mario", "Alan");
System.out.println(authors);
```

Raoul, Mario,Alan

附录 C

如何以并发方式在同一个流上执行多种操作

Java 8中，流有一个非常大的（也可能是最大的）局限性，使用时，对它操作一次仅能得到一个处理结果。实际操作中，如果你试图多次遍历同一个流，结果只有一个，那就是遭遇下面这样的异常：

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

虽然流的设计就是如此，但我们在处理流时经常希望能同时获取多个结果。譬如，你可能会用一个流来解析日志文件，就像我们在5.7.3节中所做的那样，而不是在某个单一步骤中收集多个数据。或者，你想要维持菜单的数据模型，就像我们第4章到第6章用于解释流特性的那个例子，你希望在遍历由“佳肴”构成的流时收集多种信息。

换句话说，你希望一次性向流中传递多个Lambda表达式。为了达到这一目标，你需要一个`fork`类型的方法，对每个复制的流应用不同的函数。更理想的情况是你能以并发的方式执行这些操作，用不同的线程执行各自的运算得到对应的结果。

不幸的是，这些特性目前还没有在Java 8的流实现中提供。不过，本附录会为你展示一种方法，利用一个通用API[①]，即`Spliterator`，尤其是它的延迟绑定能力，结合`BlockingQueues`和`Futures`来实现这一大有裨益的特性。

C.1 复制流

要达到在一个流上并发地执行多个操作的效果，你需要做的第一件事就是创建一个`StreamForker`，这个`StreamForker`会对原始的流进行封装，在此基础之上你可以继续定义你希望执行的各种操作。我们看看下面这段代码。

代码清单C-1 定义一个`StreamForker`，在一个流上执行多个操作

```
public class StreamForker<T> {
```

① 本附录接下来介绍的实现基于Paul Sandoz向lambda-dev邮件列表http://mail.openjdk.java.net/pipermail/lambda-dev/2013-November/011516.html提供的解决方案。

```
    private final Stream<T> stream;
    private final Map<Object, Function<Stream<T>, ?>> forks =
                                                     new HashMap<>();

    public StreamForker(Stream<T> stream) {
        this.stream = stream;
    }

    public StreamForker<T> fork(Object key, Function<Stream<T>, ?> f) {
       forks.put(key, f);
       return this;
    }

    public Results getResults() {
       // To be implemented
    }
}
```

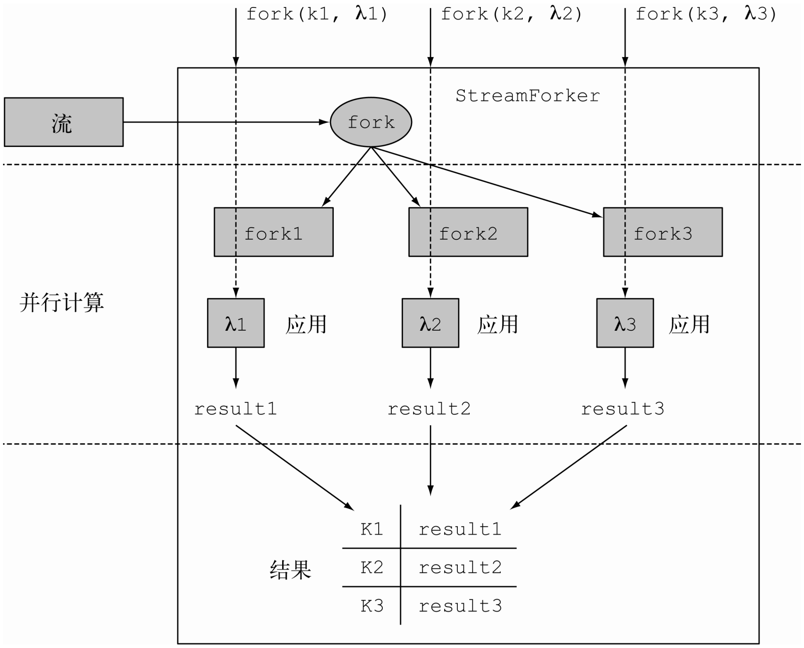
使用一个键对流上的函数进行索引

返回**this**从而保证多次流畅地调用**fork**方法

这里的fork方法接受两个参数。

- ❑ Function参数，它对流进行处理，将流转变为代表这些操作结果的任何类型。
- ❑ key参数，通过它你可以取得操作的结果，并将这些键/函数对累积到一个内部的Map中。

fork方法返回StreamForker自身，因此，你可以通过复制多个操作构造一个流水线。图C-1展示了StreamForker背后的主要思想。



图C-1　StreamForker详解

这里用户定义了希望在流上执行的三种操作，这三种操作通过三个键索引标识。StreamForker会遍历原始的流，并创建它的三个副本。这时就可以并行地在复制的流上执行这三种操作，这些函数运行的结果由对应的键进行索引，最终会填入到结果的Map。

所有由fork方法添加的操作的执行都是通过getResults方法的调用触发的，该方法返回一个Results接口的实现，具体的定义如下：

```
public static interface Results {
    public <R> R get(Object key);
}
```

这一接口只有一个方法，你可以将fork方法中使用的key对象作为参数传入，方法会返回该键对应的操作结果。

C.1.1　使用 ForkingStreamConsumer 实现 Results 接口

你可以用下面的方式实现getResults方法：

```
public Results getResults() {
    ForkingStreamConsumer<T> consumer = build();
    try {
        stream.sequential().forEach(consumer);
    } finally {
        consumer.finish();
    }
    return consumer;
}
```

ForkingStreamConsumer同时实现了前面定义的Results接口和Consumer接口。随着我们进一步剖析它的实现细节，你会看到它主要的任务就是处理流中的元素，将它们分发到多个BlockingQueues中处理，BlockingQueues的数量和通过fork方法提交的操作数是一致的。注意，我们很明确地知道流是顺序处理的，不过，如果你在一个并发流上执行forEach方法，它的元素可能就不是顺序地被插入到队列中了。finish方法会在队列的末尾插入特殊元素表明该队列已经没有更多需要处理的元素了。build方法主要用于创建ForkingStreamConsumer，详细内容请参考下面的代码清单。

代码清单C-2　使用build方法创建ForkingStreamConsumer

```
private ForkingStreamConsumer<T> build() {
    List<BlockingQueue<T>> queues = new ArrayList<>();    <-- 创建由队列组成的列表，每一个队列对应一个操作

    Map<Object, Future<?>> actions =                      <-- 建立用于标识操作的键与包含操作结果的Future之间的映射关系
            forks.entrySet().stream().reduce(
                    new HashMap<Object, Future<?>>(),
                    (map, e) -> {
                        map.put(e.getKey(),
                                getOperationResult(queues, e.getValue()));
                        return map;
                    },
```

```
                        (m1, m2) -> {
                            m1.putAll(m2);
                            return m1;
                        });
        return new ForkingStreamConsumer<>(queues, actions);
    }
```

代码清单C-2中，你首先创建了我们前面提到的由BlockingQueues组成的列表。紧接着，你创建了一个Map，Map的键就是你在流中用于标识不同操作的键，值包含在Future中，Future中包含了这些操作对应的处理结果。BlockingQueues的列表和Future组成的Map会被传递给ForkingStreamConsumer的构造函数。每个Future都是通过getOperationResult方法创建的，代码清单如下。

代码清单C-3　使用getOperationResult方法创建Future

```
private Future<?> getOperationResult(List<BlockingQueue<T>> queues,
                                     Function<Stream<T>, ?> f) {
    BlockingQueue<T> queue = new LinkedBlockingQueue<>();
    queues.add(queue);
    Spliterator<T> spliterator =new BlockingQueueSpliterator<>(queue);
    Stream<T> source = StreamSupport.stream(spliterator, false);
    return CompletableFuture.supplyAsync( () -> f.apply(source) );
}
```

创建一个队列，并将其添加到队列的列表中

创建一个Spliterator，遍历队列中的元素

创建一个流，将Spliterator作为数据源

创建一个Future对象，以异步方式计算在流上执行特定函数的结果

getOperationResult方法会创建一个新的BlockingQueue，并将其添加到队列的列表。这个队列会被传递给一个新的BlockingQueueSpliterator对象，后者是一个延迟绑定的Spliterator，它会遍历读取队列中的每个元素；我们很快会看到这是如何做到的。

接下来你创建了一个顺序流对该Spliterator进行遍历，最终你会创建一个Future在流上执行某个你希望的操作并收集其结果。这里的Future使用CompletableFuture类的一个静态工厂方法创建，CompletableFuture实现了Future接口。这是Java 8新引入的一个类，我们在第11章对它进行过详细的介绍。

C.1.2　开发 ForkingStreamConsumer 和 BlockingQueueSpliterator

还有两个非常重要的部分你需要实现，分别是前面提到过的ForkingStreamConsumer类和BlockingQueueSpliterator类。你可以用下面的方式实现前者。

代码清单C-4　实现ForkingStreamConsumer类，为其添加处理多个队列的流元素

```
static class ForkingStreamConsumer<T> implements Consumer<T>, Results {
    static final Object END_OF_STREAM = new Object();

    private final List<BlockingQueue<T>> queues;
    private final Map<Object, Future<?>> actions;

    ForkingStreamConsumer(List<BlockingQueue<T>> queues,
                          Map<Object, Future<?>> actions) {
```

```
        this.queues = queues;
        this.actions = actions;
    }

    @Override
    public void accept(T t) {
        queues.forEach(q -> q.add(t));        // 将流中遍历的元素添加到所有的队列中
    }

    void finish() {
        accept((T) END_OF_STREAM);            // 将最后一个元素添加到队列中，表明该流已经结束
    }

    @Override
    public <R> R get(Object key) {
        try {
            return ((Future<R>) actions.get(key)).get();   // 等待Future完成相关的计算，返回由特定键标识的处理结果
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

这个类同时实现了Consumer和Results接口，并持有两个引用，一个指向由BlockingQueues组成的列表，另一个是执行了由Future构成的Map结构，它们表示的是即将在流上执行的各种操作。

Consumer接口要求实现accept方法。这里，每当ForkingStreamConsumer接受流中的一个元素，它就会将该元素添加到所有的BlockingQueues中。另外，当原始流中的所有元素都添加到所有队列后，finish方法会将最后一个元素添加所有队列。BlockingQueueSpliterators碰到最后这个元素时会知道队列中不再有需要处理的元素了。

Results接口需要实现get方法。一旦处理结束，get方法会获得Map中由键索引的Future，解析处理的结果并返回。

最后，流上要进行的每个操作都会对应一个BlockingQueueSpliterator。每个BlockingQueueSpliterator都持有一个指向BlockingQueues的引用，这个BlockingQueues是由ForkingStreamConsumer生成的，你可以用下面这段代码清单类似的方法实现一个BlockingQueueSpliterator。

代码清单C-5　一个遍历BlockingQueue并读取其中元素的Spliterator

```
class BlockingQueueSpliterator<T> implements Spliterator<T> {
    private final BlockingQueue<T> q;

    BlockingQueueSpliterator(BlockingQueue<T> q) {
        this.q = q;
    }

    @Override
    public boolean tryAdvance(Consumer<? super T> action) {
```

```
            T t;
            while (true) {
                try {
                    t = q.take();
                    break;
                } catch (InterruptedException e) { }
            }

            if (t != ForkingStreamConsumer.END_OF_STREAM) {
                action.accept(t);
                return true;
            }

            return false;
        }

        @Override
        public Spliterator<T> trySplit() {
            return null;
        }

        @Override
        public long estimateSize() {
            return 0;
        }

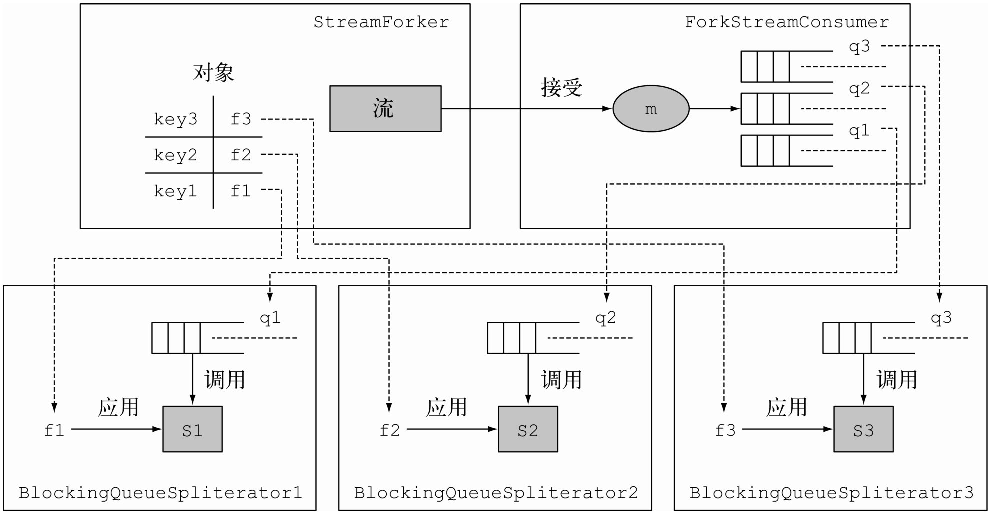
        @Override
        public int characteristics() {
            return 0;
        }
    }
```

这段代码实现了一个`Spliterator`，不过它并未定义如何切分流的策略，仅仅利用了流的延迟绑定能力。由于这个原因，它也没有实现`trySplit`方法。

由于无法预测能从队列中取得多少个元素，所以`estimatedSize`方法也无法返回任何有意义的值。更进一步，由于你没有试图进行任何切分，所以这时的估算也没什么用处。

这一实现并没有体现表7-2中列出的`Spliterator`的任何特性，因此`characteristic`方法返回`0`。

这段代码中提供了实现的唯一方法是`tryAdvance`，它从`BlockingQueue`中取得原始流中的元素，而这些元素最初由`ForkingSteamConsumer`添加。依据`getOperationResult`方法创建`Spliterator`同样的方式，这些元素会被作为进一步处理流的源头传递给`Consumer`对象（在流上要执行的函数会作为参数传递给某个`fork`方法调用）。`tryAdvance`方法返回`true`通知调用方还有其他的元素需要处理，直到它发现由`ForkingSteamConsumer`添加的特殊对象，表明队列中已经没有更多需要处理的元素了。图C-2展示了`StreamForker`及其构建模块的概述。



图C-2　StreamForker及其合作的构造块

这幅图中，左上角的StreamForker中包含一个Map结构，以方法的形式定义了流上要执行的操作，这些方法分别由对应的键索引。右边的ForkingStreamConsumer为每一种操作的对象维护了一个队列，原始流中的所有元素会被分发到这些队列中。

图的下半部分，每一个队列都有一个BlockingQueueSpliterator从队列中提取元素作为各个流处理的源头。最后，由原始流复制创建的每个流，都会被作为参数传递给某个处理函数，执行对应的操作。至此，你已经实现了StreamForker所有组件，可以开始工作了。

C.1.3　将 StreamForker 运用于实战

我们将StreamForker应用到第4章中定义的menu数据模型上，希望对它进行一些处理。通过复制原始的菜肴（dish）流，我们想以并发的方式执行四种不同的操作，代码清单如下所示。这尤其适用于以下情况：你想要生成一份由逗号分隔的菜肴名列表，计算菜单的总热量，找出热量最高的菜肴，并按照菜的类型对这些菜进行分类。

代码清单C-6　将StreamForker运用于实战

```
Stream<Dish> menuStream = menu.stream();

StreamForker.Results results = new StreamForker<Dish>(menuStream)
        .fork("shortMenu", s -> s.map(Dish::getName)
                                 .collect(joining(", ")))
        .fork("totalCalories", s -> s.mapToInt(Dish::getCalories).sum())
        .fork("mostCaloricDish", s -> s.collect(reducing(
                (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2))
                .get())
         .fork("dishesByType", s -> s.collect(groupingBy(Dish::getType)))
         .getResults();
```

```
String shortMenu = results.get("shortMenu");
int totalCalories = results.get("totalCalories");
Dish mostCaloricDish = results.get("mostCaloricDish");
Map<Dish.Type, List<Dish>> dishesByType = results.get("dishesByType");

System.out.println("Short menu: " + shortMenu);
System.out.println("Total calories: " + totalCalories);
System.out.println("Most caloric dish: " + mostCaloricDish);
System.out.println("Dishes by type: " + dishesByType);
```

StreamForker提供了一种使用简便、结构流畅的API，它能够复制流，并对每个复制的流施加不同的操作。这些应用在流上以函数的形式表示，可以用任何对象的方式标识，在这个例子里，我们选择使用String的方式。如果你没有更多的流需要添加，可以调用StreamForker的getResults方法，触发所有定义的操作开始执行，并取得StreamForker.Results。由于这些操作的内部实现就是异步的，getResults方法调用后会立刻返回，不会等待所有的操作完成，拿到所有的执行结果才返回。

你可以通过向StreamForker.Results接口传递标识特定操作的键取得某个操作的结果。如果该时刻操作已经完成，get方法会返回对应的结果；否则，该方法会阻塞，直到计算结束，取得对应的操作结果。

正如我们所预期的，这段代码会产生下面这些输出：

```
Short menu: pork, beef, chicken, french fries, rice, season fruit, pizza,
     prawns, salmon
Total calories: 4300
Most caloric dish: pork
Dishes by type: {OTHER=[french fries, rice, season fruit, pizza], MEAT=[pork,
     beef, chicken], FISH=[prawns, salmon]}
```

C.2　性能的考量

提起性能，你不应该想当然地认为这种方法比多次遍历流的方式更加高效。如果构成流的数据都保存在内存中，阻塞式队列所引发的开销很容易就抵消了由并发执行操作所带来的性能提升。

与此相反，如果操作涉及大量的I/O，譬如流的源头是一个巨型文件，那么单次访问流可能是个不错的选择；因此（大多数情况下）优化应用性能唯一有意义的规则是“好好地度量它”。

通过这个例子，我们展示了怎样一次性地在同一个流上执行多个操作。更重要地是，我们相信这个例子也证明了一点，即使某个特性原生的Java API暂时还不支持，充分利用Lambda表达式的灵活性和一点点的创意，整合现有的功能，你完全可以实现想要的新特性。

附录 D

Lambda表达式和JVM字节码

你可能会好奇Java编译器是如何实现Lambda表达式，而Java虚拟机又是如何对它们进行处理的。如果你认为Lambda表达式就是简单地被转换为匿名类，那就太天真了，请继续阅读下去。本附录通过审视编译生成的.class文件，简要地讨论Java是如何编译Lambda表达式的。

D.1 匿名类

我们在第2章已经介绍过，匿名类可以同时声明和实例化一个类。因此，它们和Lambda表达式一样，也能用于提供函数式接口的实现。

由于Lambda表达式提供了函数式接口中抽象方法的实现，这让人有一种感觉，似乎在编译过程中让Java编译器直接将Lambda表达式转换为匿名类更直观。不过，匿名类有着种种不尽如人意的特性，会对应用程序的性能带来负面影响。

- 编译器会为每个匿名类生成一个新的.class文件。这些新生成的类文件的文件名通常以`ClassName$1`这种形式呈现，其中`ClassName`是匿名类出现的类的名字，紧跟着一个美元符号和一个数字。生成大量的类文件是不利的，因为每个类文件在使用之前都需要加载和验证，这会直接影响应用的启动性能。如果将Lambda表达式转换为匿名类，每个Lambda表达式都会产生一个新的类文件，这是我们不期望发生的。
- 每个新的匿名类都会为类或者接口产生一个新的子类型。如果你为了实现一个比较器，使用了一百多个不同的Lambda表达式，这意味着该比较器会有一百多个不同的子类型。这种情况下，JVM的运行时性能调优会变得更加困难。

D.2 生成字节码

Java的源代码文件会经由Java编译器编译为Java字节码。之后JVM可以执行这些生成的字节码运行应用。编译时，匿名类和Lambda表达式使用了不同的字节码指令。你可以通过下面这条命令查看任何类文件的字节码和常量池：

```
javap -c -v ClassName
```

我们试着使用Java 7中旧的格式实现了Function接口的一个实例，代码如下所示。

代码清单D-1　以匿名内部类的方式实现的一个Function接口

```
import java.util.function.Function;
public class InnerClass {
    Function<Object, String> f = new Function<Object, String>() {
        @Override
        public String apply(Object obj) {
            return obj.toString();
        }
    };
}
```

这种方式下，和Function对应，以匿名内部类形式生成的字节码看起来就像下面这样：

```
 0: aload_0
 1: invokespecial #1       // Method java/lang/Object."<init>":()V
 4: aload_0
 5: new           #2       // class InnerClass$1
 8: dup
 9: aload_0
10: invokespecial #3       // Method InnerClass$1."<init>":(LInnerClass;)V
13: putfield      #4       // Field f:Ljava/util/function/Function;
16: return
```

这段代码展示了下面这些编译中的细节。

- 通过字节码操作new，一个InnerClass$1类型的对象被实例化了。与此同时，一个指向新创建对象的引用会被压入栈。
- dup操作会复制栈上的引用。
- 接着，这个值会被invokespecial指令处理，该指令会初始化对象。
- 栈顶现在包含了指向对象的引用，该值通过putfield指令保存到了LambdaBytecode类的f1字段。

InnerClass$1是由编译器为匿名类生成的名字。如果你想要再次确认这一情况，也可以查看InnerClass$1类文件，你可以看到Function接口的实现代码如下：

```
class InnerClass$1 implements
          java.util.function.Function<java.lang.Object, java.lang.String> {
  final InnerClass this$0;
  public java.lang.String apply(java.lang.Object);
    Code:
       0: aload_1
       1: invokevirtual #3 //Method
                              java/lang/Object.toString:()Ljava/lang/String;
       4: areturn
}
```

D.3　用 InvokeDynamic 力挽狂澜

现在，我们试着采用Java 8中新提供的Lambda表达式来完成同样的功能。我们会查看下面这段代码清单生成的类文件。

代码清单D-2　使用Lambda表达式实现的Function

```
import java.util.function.Function;
public class Lambda {
    Function<Object, String> f = obj -> obj.toString();
}
```

你会看到下面这些字节码指令：

```
 0: aload_0
 1: invokespecial #1    // Method java/lang/Object."<init>":()V
 4: aload_0
 5: invokedynamic #2, 0 // InvokeDynamic
                           #0:apply:()Ljava/util/function/Function;
10: putfield      #3    // Field f:Ljava/util/function/Function;
13: return
```

我们已经解释过将Lambda表达式转换为内部匿名类的缺点，通过这段字节码你可以再次确认二者之间巨大的差别。创建额外的类现在被invokedynamic指令替代了。

invokedynamic指令

字节码指令invokedynamic最初被JDK7引入，用于支持运行于JVM上的动态类型语言。执行方法调用时，invokedynamic添加了更高层的抽象，使得一部分逻辑可以依据动态语言的特征来决定调用目标。这一指令的典型使用场景如下：

```
def add(a, b) { a + b }
```

这里a和b的类型在编译时都未知，有可能随着运行时发生变化。由于这个原因，JVM首次执行invokedynamic调用时，它会查询一个bootstrap方法，该方法实现了依赖语言的逻辑，可以决定选择哪一个方法进行调用。bootstrap方法返回一个链接调用点(linked call site)。很多情况下，如果add方法使用两个int类型的变量，紧接下来的调用也会使用两个int类型的值。所以，每次调用也没有必要都重新选择调用的方法。调用点自身就包含了一定的逻辑，可以判断在什么情况下需要进行重新链接。

代码清单D-2中，使用invokedynamic指令的目的略微有别于我们最初介绍的那一种。这个例子中，它被用于延迟Lambda表达式到字节码的转换，最终这一操作被推迟到了运行时。换句话说，以这种方式使用invokedynamic，可以将实现Lambda表达式的这部分代码的字节码生成推迟到运行时。这种设计选择带来了一系列好结果。

- Lambda表达式的代码块到字节码的转换由高层的策略变成了纯粹的实现细节。它现在可以动态地改变，或者在未来版本中得到优化、修改，并且保持了字节码的后向兼容性。

- 没有带来额外的开销，没有额外的字段，也不需要进行静态初始化，而这些如果不使用Lambda，就不会实现。
- 对无状态非捕获型Lambda，我们可以创建一个Lambda对象的实例，对其进行缓存，之后对同一对象的访问都返回同样的内容。这是一种常见的用例，也是人们在Java 8之前就惯用的方式；比如，以static final变量的方式声明某个比较器实例。
- 没有额外的性能开销，因为这些转换都是必须的，并且结果也进行了链接，仅在Lambda首次被调用时需要转换。其后所有的调用都能直接跳过这一步，直接调用之前链接的实现。

D.4　代码生成策略

将Lambda表达式的代码体填入到运行时动态创建的静态方法，就完成了Lambda表达式的字节码转换。无状态Lambda在它涵盖的范围内不保持任何状态信息，就像我们在代码清单D-2中定义的那样，字节码转换时它是所有Lambda中最简单的一种类型。这种情况下，编译器可以生成一个方法，该方法含有该Lambda表达式同样的签名，所以最终转换的结果从逻辑上看起来就像下面这样：

```
public class Lambda {
    Function<Object, String> f = [dynamic invocation of lambda$1]

    static String lambda$1(Object obj) {
        return obj.toString();
    }
}
```

Lambda表达式中包含了final（或者效果上等同于final）的本地变量或者字段的情况会稍微复杂一些，就像下面的这个例子：

```
public class Lambda {
    String header = "This is a ";
    Function<Object, String> f = obj -> header + obj.toString();
}
```

这个例子中，生成方法的签名不会和Lambda表达式一样，因为它还需要携带参数来传递上下文中额外的状态。为了实现这一目标，最简单的方案是在Lambda表达式中为每一个需要额外保存的变量预留参数，所以实现前面Lambda表达式的生成方法会像下面这样：

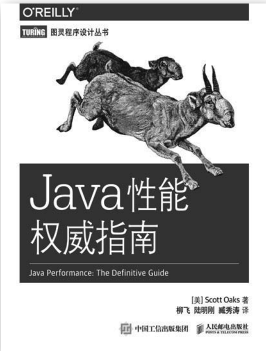
```
public class Lambda {
    String header = "This is a ";
    Function<Object, String> f = [dynamic invocation of lambda$1]

    static String lambda$1(String header, Object obj) {
        return obj -> header + obj.toString();
    }
}
```

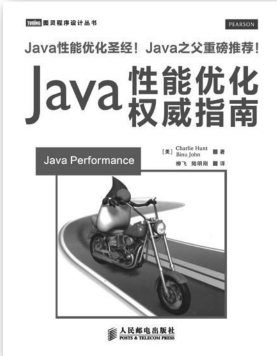
更多关于Lambda表达式转换流程的内容，可以访问如下地址：http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html。

TURING 图灵教育

延 展 阅 读



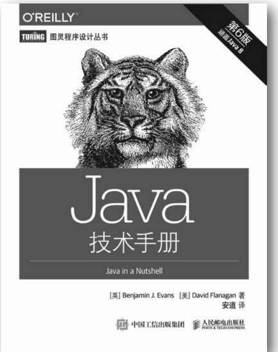
书号：978-7-115-41376-5
定价：79.00 元



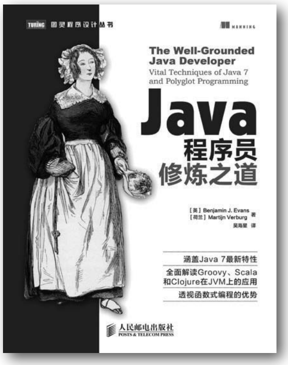
书号：978-7-115-34297-3
定价：109.00 元



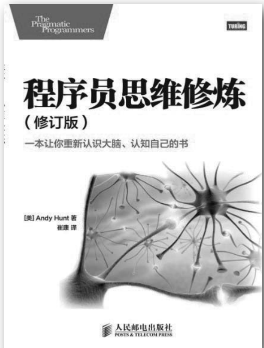
书号：978-7-115-38488-1
定价：39.00 元



书号：978-7-115-40609-5
定价：79.00 元



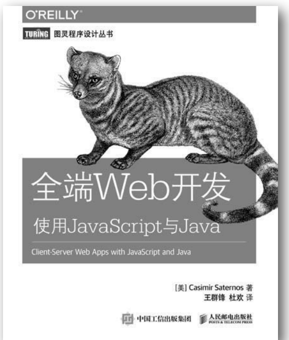
书号：978-7-115-32195-4
定价：89.00 元



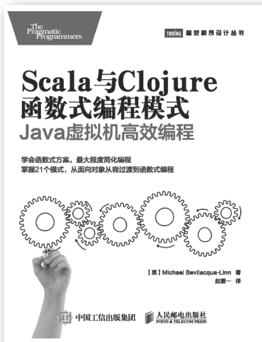
书号：978-7-115-37493-6
定价：49.00 元



书号：978-7-115-39130-8
定价：49.00 元



书号：978-7-115-39730-0
定价：59.00 元



书号：978-7-115-38894-0
定价：49.00 元

Java 8的发布使Java程序设计发生了翻天覆地的变化。利用Java 8中新引入的函数式特性，你可以在更短的时间内用更简洁的代码完成更复杂的功能，同时还能充分利用硬件的多核架构。

本书结构清晰、内容翔实，从实例入手，涵盖Java 8的主要新特性，包括Lambda表达式、方法引用、流、默认方法、`Optional`、`CompletableFuture`以及新的日期和时间API，是程序员了解Java 8新特性的终极指南。

本书的主要内容如下：

- 如何使用Java 8新增的强大特性
- 如何编写能有效利用多核架构的程序
- 重构、测试和调试
- 怎样高效地应用函数式编程

"这是一部十分优秀且简明的著作，书中提供了大量的示例，能帮助你迅速地掌握Java 8中的新特性。"

——Jason Lee，甲骨文公司

"这是最棒的Java 8指南！"

——William Wheeler，ProData Computer Systems公司

"这本书中有关新的Stream API及Lambda表达式的示例非常有用。"

——Steve Rogers，CGTek公司

"这是所有使用Java 8函数式特性的程序员都必备的工具书。"

——Mayur S. Patil，麻省理工学院

MANNING

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-41934-7

ISBN 978-7-115-41934-7

定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks